1990

Trek: a real time multi-player game for Xerox networked workstations

John J. Kemp

Rochester Institute of Technology
School of Computer Science and Information Technology

Trek

a real time multi-player game
for Xerox networked workstations

by
John J. Kemp

A thesis, submitted to
The Faculty of the School of Computer Science and Information Technology.

Approved by:

Prof. James Heliotis Chairman

Prof. Andrew Kitchen Committee Member

Prof. Peter Anderson Dept. Chairman

July 31, 1990

Trek - a real time multi-player game for Xerox networked workstations. I John J. Kemp hereby grant permission to the Wallace memorial library of R.I.T. to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

John J. Kemp
September 17, 1990

Table of Contents

Figures and Tables

1.0 Objective

The objective of my thesis was to implement Trek, a multi-player real time space combat game, on the Xerox 8010 and 6085 Workstations. The goal of the game is for a player to place one or more of his starbases in all of the solar systems throughout the galaxy, and to prevent others from achieving this goal.

2.0 Introduction

Over the years, many space conflict computer games have been developed. One of the more interesting and complex games was Trek, developed for the Xerox Alto workstation. What made it interesting and complex was that in the early days of computer networking, it was played over a 3 megabit ethernet with each workstation acting as an independent ship in space. Player commands were accepted, other participants notified of state changes, and each workstation updated its graphic display data appropriately. My thesis proposal was to create a similar Trek game for the newer Xerox workstations (8010 & 6085), on the 10 megabit ethernet. This involved accessing the ethernet at the lowest possible level to expedite communications, effective interaction with workstation graphics to provide a pleasant and effective user interface, and efficient / speedy software to control the game and compute ship movements through space for display and transmission.

2.1 Trademarks

XDE, 8010, 6085, XNS and Ethernet are trademarks of Xerox Corporation.

2.2 History and Evolution

The original Trek game was written in BCPL by Eugene Ball [2]. BCPL is a general purpose recursive programming language for systems programming applications[3]. Trek utilized the experimental 3 megabit ethernet and the PUP protocol developed by Xerox in the early 70's. It was played with each workstation acting as an independent ship in space. However, it utilized the entire machine's resources during execution.

The new version is written in Mesa for the Xerox Development Environment (XDE) on the newer Xerox workstations, 6085 and 8010. It utilizes the 10 megabit ethernet and the XNS protocol. The

game runs in a window in the workstation's multi-window interface, coexisting with other workstation processes.

2.3 Summary of related work

The following sections discuss games similar to my game. All are real time multi-player networked games. Three of them exist or are under development for various environments on the Xerox workstations (6085 and 8010).

2.3.1 Original Trek

Many arcade and computer games have been based on the Star Trek series. Most have only allowed randomly generated or computer controlled ships to compete. In this game, each ship is another player under human control. His movements and actions are transmitted to all other players in real time. This game is the father of all the games in this section. Its development, in the early days of networking, sparked the idea for the other games development. The details of the game will be described in section 3 as background for my proposed project.

2.3.2 Mazewar

2.3.2.1 Game

A game similar to Trek in its use of the net for real time combat is Mazewar. Mazewar, by Bryan Yamamoto [12] was developed for the Alto in Mesa, and rehosted to the new workstations. It runs in both XDE and Viewpoint. The game uses a maze as its field of battle. Players are shown a display of their pieces field of view, a global view of the maze with their piece in it, and a list of players with their scores. They are not shown where their opponents are, unless they are in the field of view. The player's field of view displays what is directly in front of them or what they see when they peek left or right around corners. Opponents appear to them as eye balls, at which they then shoot. Points are

received for hitting an opponent , and taken away for being hit. When hit, your eye is removed and transported to a new random location in the maze to continue play.

2.3.2.2 Net Communications

Mazewar, on the 6085 & 8010, uses XNS communications through Socket, a non-public communications interface. This interface allows the game to send and receive packets to and from a given network address. The network address used by the game is comprised of the game's well-known socket, number 253, the broadcast host number, and typically the local net number. Mazewar does however allow a player to specify the local net on which he wishes to play, using the internet to facilitate communications. The number of players in Mazewar is limited to eight. I believe this is to limit packet processing delays in the local machine, but may be used to reduce network loading.

2.3.3 Tank

2.3.3.1 The Game

Taken from an arcade game by the same name, Tank by Russ Atkinson [1] [7] is a real time multi-player game written and played in Cedar. It runs on an augmented Xerox workstation, in the Cedar environment. Cedar is an environment similar to XDE, but with added functionality. Similarly Cedar, the language, is a super set of Mesa. In this game players run their own tank and their display is the view out the front of the tank. The tank is able to turn, move, and shoot. It plays on an enclosed field, with obstacles and anti-tank mines. Players attempt to destroy other players' tanks while avoiding the mines, dodging tank projectiles, and hiding behind obstacles.

2.3.3.2 Net Communications

Unlike Trek and Mazewar, Tank uses a server to maintain a consistent view of the game world. When the game is started one of the participating player's machines is selected randomly to be the game's server. Each participating machine periodically initiates an update session in which it

transmits a packet to the server, stating its position and that of its projectile. In response, the server transmits a packet with the state of the world. If a player's tank believes it has scored a hit, it sends a message to that effect to the server. The server records this information, but does not validate it.

2.3.4 Blub

2.3.4.1 The Game

Like my game, Trek,Blub is a multi-player real time game played in XDE, on the Xerox workstation. It is based on the game Gato. The idea is to travel around the world, sink enemy ships, and protect your own. [4][5][6] It is currently being developed by Thomas Jell and Gerhard Henning at Siemans Corporation in Munich, Germany. The game is set in a sea with islands. The ships available vary from freighters to aircraft carriers and submarines. There are two teams in Blub. Players are assigned to the team with the fewest players when they enter the game or given their choice when the teams are equal. Players start out commanding a freighter. Successful completion of a mission yields points. As the player's points increase, the class of ship he may command is increased. A mission varies depending on the ship class. For a freighter, the mission is delivery of goods. For a submarine, it is sinking enemy ships. These were the first ships implemented others are being developed. Ultimately, when a player returns to his home port his ship is resupplied and he is offered the opportunity to change ship class based on his accumulated points. Currently however, the player is allowed to select any of the operating ship classes for test, regardless of points.

Unlike my game, but like the old Trek, the primary user interface is comprised of one large window (13 x 11). It is however a citizen of the Tajo window system and can be closed or manipulated about the screen. The window is comprised of five subwindows. The top subwindow is a message subwindow for the game to communicate with the player. Below it is a subwindow divided horizontally into four sections, the left three being for the radio and the right most being a command subwindow for terminating play, requesting help, and listing the current scores. The three

radio subwindows are from left to right for posting incoming messages, control (setting the frequency, on/off, and SOS), and sending a message. The center subwindow is planned to be split into two parts. A local surface visibility display to the left and a radar display to the right. The radar display is currently not implemented. One up from the bottom is a subwindow which is split into many sub parts. This subwindow varies with the ship being presented. It has three analog displays for speed, heading, and RPM or depth; surface ships don't need depth and submarines do without RPM. It also has rudder and engine speed controls here. For submarines bow angle and ballast controls are also presented. Engine selection is provided in another area of this subwindow. For most ships this is just activation of diesel engines and status of fuel supply. However, for submarines electric engines are also available along with battery charge status and charging information. The captain is responsible for switching engines when he dives. The remaining areas of the subwindow provide damage information, score, ship alliance, and team score. The bottom subwindow is a log. It initially contains information about the ship selected. Help, ship mission, and score requests are printed here. One other window is available. It too is 13 x 11. This is the chart and is activated by a command in the fourth subwindow. The chart provides a global view of the sea, its islands and where you are in it. It takes extra effort to provide this display so play is more responsive if it is not shown.

2.3.4.2 Net Communications

Like Tank, Blub uses a server to centralize its communications. This server however is typically a separate machine. Though it may be a player's machine, this is not recommended. Each time the server is started it randomly creates a new set of islands in its sea. It also provides a minimum of two system controlled freighters running at all times from port to port. When a player starts up he specifies which server he would like to connect to. This chooses a game but is not necessarily the server the player finally connects with. Blub then connects him with a server and play may proceed even across the internet. This has been shown to work well up to three hops away. However, if more than four ships are playing, it is recommended that auxiliary servers be set up. These servers

communicate with the main server for global ship and island position information and off load direct workstation communications. Servers are notified of ship course and speed changes. The main server utilizes a real time clock to plot ship positions. It reports this information to auxiliary servers if there are any. The main or auxiliary servers then report to participating workstations with information on visible islands and ships. The workstation then clips and displays this information visually along with local ship status information.

3.0 Background - Alto Trek

In order to understand the game I developed it is necessary to understand the original game. The original Trek was developed by Eugene Ball and a number of collaborators [2]. It ran on the Alto workstation. The following sections will describe how the game was run and how it was presented to the user.

3.1 Starting the game

The game was initially run by an executive command of trek. Initial options could be supplied as switches in the run line, or the player would be prompted for them. The options included race, solar system, gravity, battle mode, and number of clusters.

3.1.1 Race

The race option allowed the individual to choose the type of ship he wanted; Terran, Klingon, or Romulan. Terran ships started with 32 torpedoes and 4 star bases. They fired 3 medium range phaser streaks and reloaded their torpedoes at a moderate rate. The Klingons started with 15 torpedoes and 3 star bases. They fired only one long phaser streak but reloaded torpedoes rapidly. The Romulans began with 45 torpedoes and 5 star bases. They fired 3 short widely spaced phaser streaks and were slow to reload their torpedoes. The Romulans possessed another option which the other races did not have. They had a cloaking device which prevented them from seeing and from being seen. When activated, the players ship disappeared from all other ships' long range scan displays, but remain visible in short range. The activating player's short range long would go blank, but his short range scan would still work.

3.1.2 Solar System

The solar system option allowed the player to determine which system he should start in. This was convenient for players who wished to have a private dual. They could agree upon the solar system and then begin play without being bothered by other players who typically begin play in the system named Sol.

3.1.3 Gravity

The gravity option switched gravity on or off. Gravity, as the name implies, allows mass to attract mass. Playing with gravity off was like using automatic stabilizers to keep the players ship on course. Playing with gravity on put total control in the players hands, allowing the ship to drift into planets, star bases, and other ships, if the captain was not careful.

3.1.4 Battle Mode

Battle mode was an option which first set the player up for combat. This code switched on the battle computer, and placed the ship in Sol. Sol is where most general combat was held. Entry to the game was random, within the selected solar system. This means that a new entry could be placed in the middle of several ships engaged in heavy fighting. The battle computer, if switched on, would automatically put up shields and refresh them as necessary. It also would target phasers and torpedoes at the closest ship. Entering without the battle computer could leave the ship defenseless in a hostile environment with the player scrambling to raise shields and target aggressor ships.

Once in the game, use of the Battle Computer required less skill than not using it. It would target weapons, direct the ship, and operate the shields. The player need only accelerate the ship and fire the weapons. At one time it even did this, but due to lack of player skill required these features were removed. Because the battle computer targets the closest ship, tactics and strategy were limited by its

use. If players were playing as teams, the battle computer would not know about it and in targeting the closest ship it might target a fellow team member. Further, the run away option was removed by the battle computer. This was a result of the battle computer targeting torpedoes, which in turn changes the ships direction. Thus when accelerating to run away, the battle computer would redirect the ship back into the fray.

3.1.5 Clusters

The number of clusters option actually sets up the number of solar systems in the player's galaxy. The maximum and default value was 15. Numbers lower than that removed solar systems from the galaxy, yielding a subset of the default. This made it easier to accomplish the game's goal of placing a star base in each system by having fewer systems to reach. In the extreme case, by setting this to one, a player could enter Sol, drop a star base, and be declared master of the Universe and receive other ego lifting compliments.

3.2 The User Interface

Figure 3.1 shows a graphic representation of the Alto Trek display. It consumed all of the 8.5 × 11 inch display. The display contains long and short range scans, weapons, velocity, and damage status fields, as well as attitude, speed and fire control fields. It also offered a galactic map to be displayed in place of the long range scan, which is not shown in this reproduction.

3.2.1 Short Range Scan, Long Range Scan, and Galactic Map

The upper left corner contained the long range scan and Galactic map, with the bottom half of the screen showing the short range scan. The galactic map was displayed by clicking any mouse button when the cursor was in the long range scan. The long range scan was returned to by clicking the right mouse button with the cursor in the Galactic map. The Galactic map displayed the solar systems in the games universe. It highlighted the solar system presently containing the player's ship

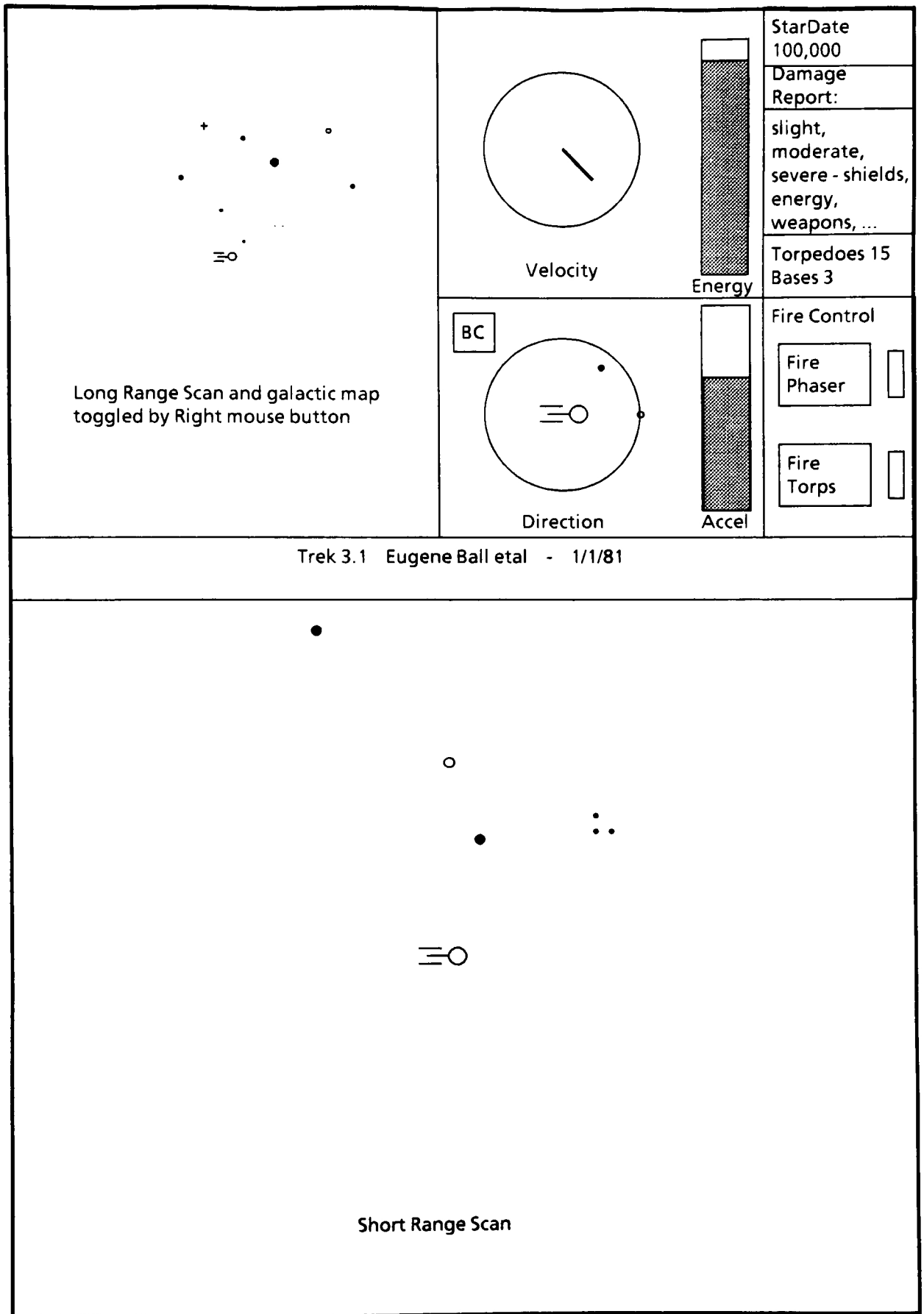


Figure 3.1 Alto Trek Display Simulation.

and bolded those which had star bases belonging to the player. The long range scan showed a large area of the solar system around the ship in miniature. The short range scan showed the center of the long range scan, in large scale. It also showed details not in the long range scan such as phaser and torpedo fire.

3.2.2 Status Information

The upper right hand corner was divided into two major areas, status and control. The status information was in the top half. It had a velocity vector showing ship course and speed. The circle around the velocity vector indicated a maximum speed of one, which was warp speed. The ship however could exceed this speed, but the indicator could not show it. An Energy histogram was supplied to indicate energy consumption and resupply. If depleted, life support would fail and the player would die. The stardate field began at 100,000 and incremented every second. It could be used to see how long the game had been played without death. The damage report field listed damage as it occurred. Damage would be given qualifiers such as slight, moderate, and severe. Things which could be damaged were attitude control, phasers, torpedoes, shields, hyper drive, energy systems, impulse drive, and more. As repairs were made automatically the qualifiers would lessen and eventually the report would go away. Aside from the report, items which were damaged would be taken away. For example, loss of energy systems would stop the replenishment of energy. If shields, weapons, and engines were allowed to continue to take energy, the ship would perish. Damage to torpedoes would remove the capability of firing them and so on. The bottom right status window displayed the number of torpedoes and bases on board. When you ran out of torpedoes you could not shoot any more, and when you ran out of bases you could not drop any more. Both of these resources could only be replenished by docking with a star base.

3.2.3 Fire and Directional Control

Below the status fields in the lower half of the upper right corner were the control fields. The fire control section could be clicked with any button over the desired weapon to fire it. The Acceleration histogram could be set with the left or center mouse button for continuous acceleration. The right mouse button could be used to set acceleration which would gradually drop to a constant speed maintaining acceleration. In either case, when warp speed was reached the acceleration would drop to 0. The BC square represented a boolean. When highlighted the battle computer was switched on, otherwise it was off. This could be toggled by any mouse button. The direction circle is by far the most complex region. The ship was directed by the dot on the outer circle. The dot was positioned by the left mouse button being clicked at the desired spot on the circle. Inside the circle immediately around the ship, shields could be raised and lowered by the mouse. The left and center buttons raised them and the right button lowered them. They had 5 settings (off, light, low medium, high medium, and high). The mouse buttons stepped up and down through each of these. The region between the shields and the circle is where fire control was principally directed. Another dot existed here to target the ship's phasers. The left mouse button positioned the phaser directional, the center button fired the phasers, and the right button fired the torpedoes. The torpedoes were fired in the ships forward direction at the moment of firing.

3.3 Star Bases

Star bases have been mentioned several times thus far but not defined. Their function was to build supplies and more star bases. By placing them near asteroids, they could accomplish this quicker. They were also the key to winning the game by one or more being placed in all solar systems in the player's galaxy. When a player's ship was damaged or depleted he would limp back to one of his star bases to be repaired and replenished. Torpedoes, bases, and energy would be replaced to

maximum amounts, or to what ever level the base was able to supply. What the base was able to supply was dependent on its length of operation and its location.

Star bases also supplied remote functions like ship transporters, described in the next section, remote scans and destruct, described here. Remote scans allowed a player to select a system in his galactic map and have a star base in that system provide a long range scan. This was often helpful when the ships own scanners were damaged and inoperative. The player could drop a base, or make use of an existing one, to receive a remote scan of the local system. This would allow him to identify any potential threats he otherwise could not have seen. The scan could also be used to examine a remote system when looking for an opponent. The destruct option allowed a player to use the star base as a mine. While viewing through the base, if other ships were to approach it, the base could be destroyed. This would destroy or damage all ships in proximity to the exploding base.

3.4 Warp Points and Transporters

Traveling between solar systems has been mentioned but how to do it has not. Warp Points are one of the means by which this was achieved. Warp Points were represented by a + on the scan displays. A ship passing through the + at warp speed would jump to the solar system closest to the current direction of travel. The other way to travel between solar systems or jump within a system was to use a star base transporter. A ship could use a star base which it had previously dropped in any system to transport the ship to the base.

3.5 Ship Communications

The final User Interface feature not yet discussed is ship message transmissions. Each ship was able to enter and broadcast messages to other players. The message was entered by typing on the keyboard. It was sent by hitting return. Other ships received the message and it was displayed in the center strip of the window interface, beneath the authors name.

4.0 Trek Specification - New Trek

The new game is slightly different from the original. This is an outgrowth of hardware feature changes, and enhancements I wanted to see. Hardware charges are most notably the variable display sizes and a two button instead of three button mouse. Other changes such as a higher bandwidth net, faster processor, and larger memory affect implementation issues, but are not as noticeable to the user. The following discusses changes to accommodate the display, mouse, and enhancements I made. Aside from these, the game remains essentially unchanged.

4.1 Starting the game

I have the game run in XDE, rather than its own world, as old Trek was. The game's start up is from the XDE executive window by the command trek. Option setting is through a command window.

4.2 New User Interface

The most important principle in the development environment is that users should have complete control over their environment [9] . To accommodate this and the new variable display sizes (6085, 15" or 19" and 8014, 17"), the tool should reside in windows whose size, shape, and visibility may be set by the player, thus allowing him to maximize his display's real estate, and maintain control of his environment. I implemented the user interface as a collection of small windows, rather than one large one, to meet these requirements.

4.2.1 Windows

The windows are set up to provide short range scan, long range scan, galactic map, ship status information, and ship control each in its own window. This places the functional areas of Alto Treks display in separate windows, and breaks out the galactic map into a separate window.

4.2.2 Fire Control

To accommodate the two button versus three button mouse, controls needed to be slightly different. The best solution was to use the left button to position the phaser and torpedo fire control directionals. The right button is then used for firing the phasers and torpedoes in their direction positioning regions. This eliminates the need for the center button in the phaser directional area by moving torpedo fire control to the torpedo targeting region, and moving the phaser fire control to the right button. The torpedo directional is the ship directional, as torpedoes will still fire in the direction the ship is facing.

4.3 Warp Points and Transporters

I believe warp points were created to prevent warp jumps at undesired times simply because the ship hit warp speed. It would be annoying to recover from such an accident, so I believe the developers added a requirement of a specific location in space to reduce the probability of such accidents. An enhancement I have made is the removal of warp points. Warp jumps are made when the ship reaches maximum speed with the warp engines on, not only if it reaches warp speed at a specific point in space. My solution to accidental warp jumps is the warp engines. The old game mentioned them only as something which could be lost due to damage. When damaged the ship could not make warp jumps. I felt that a boolean should be added to the controls for warp engines on or off. When off, the ship is on impulse only, and restricted to sub light speeds. When on, if the ship reaches maximum speed, the warp engines kick in and the ship jumps to the next solar system. This allows an escape mechanism similar to that made possible by star base transporters which are also implemented. To prevent ships from abusing these mechanisms by activating the transporter or running around at maximum speed and popping on the warp engines to disappear at the first sign of danger, a time delay of three seconds was imposed between warp speed and actual entry to warp,

the last second of which is spent without shields. This can be seen as the time for the shields to be lowered and the warp engines to engage.

4.4 Ship Communications

The message window has a string field to handle ship communications. The message is entered directly in the field by selecting and typing there. The message is sent by selecting the send command or typing a carriage return. It is broadcast to all players in all solar systems. This allows players to notify one another of their existence and location.

4.5 Command Entry

The keyboard is only used for message transmission, and setting initial game parameters, unlike the old game, which used keyboard escape characters to cloak, drop bases, refresh the screen, and quit the game. These operations have been added to the command window. This makes the game less cryptic by having all available options at the click of the mouse.

5.0 Implementation

This section discusses how the new game is implemented. It specifies internal and network data flow as well as the display layout and damage assessment.

5.1 Software Data Flow

A single record describes the heart of the software architecture containing pointers to all vital pieces of information. It is allocated from a heap with most expected storage at game start up. A pointer to this record is passed to procedures needing the global data elements. From this record, the local solar system and information on all ships and torpedoes in it can be found. Using the system to index into the Universe array, the fixed bodies of the solar system may be found. The Universe array, setup when the game is loaded, is constant for all ships.

5.2 Ethernet Information Passing

The Packet Exchange Interface [10] provided by Pilot is used for ethernet information passing. It allows access to the ethernet at the Transport layer with datagram service, and is designed for two way service communications. Each workstation playing the game is set up as a requestor and a replier. As a replier, the ship accepts requests from requestors and may send a response. As a requestor, the ship sends out information which may or may not receive a reply or acknowledgement. For my purposes, all packets are broadcasts or directed messages without acknowledgement.

Originally one socket, filtered on the receiving ships local solar system, was considered for Trek communications. Then two sockets were planned. One, a System socket, for handling Solar system specific communications; these were primarily local periodic ship status transmissions. The other, a Universe socket, for handling Universe wide broadcasts. This included intergalactic messages and special case acknowledged exchanges. Finally, the issue of star base control entered the scenario and

I returned to one socket, number 1015, handling Universe wide communications. The optimal way to handle bases is through a status broadcast as was originally planned for ships. However, as base information is necessary for all systems, not just the one that the ship is in, this data must be sent to the universe. To save on overhead, by having the sender build only one packet per period and thus the receiver only parse one per ship, I have decided to combine the ship status and base status packets together. The cost of this is 12 bytes of information added to the base status packet, indicating the sending ship's system, position, speed, direction, and heading. The following sections define the various packet types used and flow of information through them. The packets are defined as bytes in 16 bit words, which is how the hardware sees them. The software however views them as byte streams and is unconcerned with word boundaries except where they work to its advantage. Garbage bytes, to fill odd byte length packets, are ignored. Each packet is prefixed with a 16 bit version number and an 8 bit packet type identifier. This allows immediate inspection, upon receipt of a packet, as to whether it is valid and what to do with it.

5.2.1 Ship and Base Status Packet

5.2.1.1 Packet Layout

During the initial design phase, only variable size status packets were considered. This would allow an arbitrary number of bases deposited in what ever systems the player desired and sacrificed some decomposition speed for size, storage, and bandwidth. However, when considering implementation, it was realized that a maximum packet length needed to be set for Packet Exchange and thus I began to consider a static size packet layout. Static packets have an advantage over dynamic packets in speed for accessing a packet's content, but sacrifice size, storage, and bandwidth in the process. To realize a fixed size some limitation must be placed on the number of bases a player is allowed. I have chosen to follow the original trek games choice and limit the number of bases per system rather than the number of bases a player has in the universe. I differ from the original game in

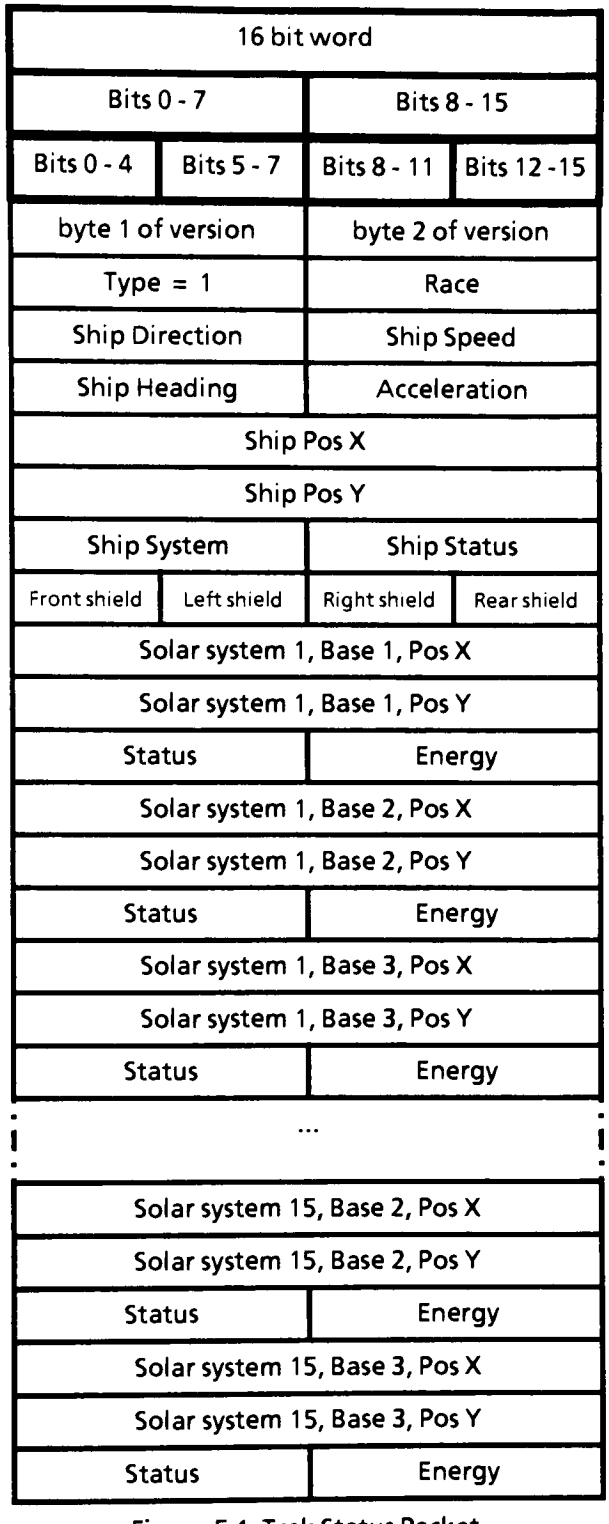


Figure 5.1 Trek Status Packet.
192 bytes of data.
Largest packet sent or received.

that I allow three bases per system rather than a maximum of two bases as in the old game. Figure 5.1 shows the status packet layout of choice.

5.2.1.2 Packet Use

Each ship will periodically broadcast status packets. The period is once per second. This is reasonably short so that ship movements are not to jittery. It is however long enough that status packet processing only consumes 5.6% of the workstation's compute cycles. The game itself consumes about 75% of the cpu. I have tried faster rates, but the cpu is pushed to 100% at .75 second intervals. This actually works until you do something. From then on, the game struggles to catch up.

The packet contains all pertinent ship and base status information for the sender. Ship information is used by ships sharing the same ship system value. Base information is inspected by all ships. Each ship extracts information for the base system that matches its local system.

The ship's position information is compared to the local ship's position and clipped to its displays. The position information is also used by the receiving ship's battle computer. If switched on and functional, this information targets torpedo and phaser fire at the closest enemy ship. Direction and speed form the ships velocity vector. Heading and acceleration create its acceleration vector. The heading value is also used to set the ships angle on the display.

These packets mark a ship's entry and exit from a solar system, its status in the system, and supply its universal base information. Entry to a solar system can be as a result of beginning play, being recreated after destruction, or warping in from another solar system. Departure can be as a result of being destroyed, warping out of another solar system, or ending the game. Entry to a system is denoted by the system value field. When it matches a receiving ship's local system that ship begins to record the packets ship data. Exit from a system of the game is denoted by the ship status field, which

is set to delete when the ship is to be removed from a system. The ship status field is also used to indicate when a ship is cloaked.

As ship positions are transmitted periodically, all that is necessary to learn the state of a solar system is to wait the periodic interval and receive all current ship broadcast positions. Failure to receive a ships packets for three seconds marks it as dead. It and its bases are removed from the game. This prevents ship bodies from lying around the galaxy if a machine crashes or boots out of the game. This period is long enough that deletion will not occur as a result of a communications failure. If a ship is deleted, but not really gone, when its next status packet are received, it and its bases are reentered in each ships data structures as if they were never gone, but merely transparent for a period of time.

The frequent transmission of ship positions did not result in a heavy load on the ethernet, from many short packets being sent frequently. Therefore it was not desirable for ships to broadcast course changes instead of position, or to broadcast position and course information less frequently, computing intermediate positions. In the course change only scenario, no communication from a ship only means it is idle. Therefore it is not easy to detect that a ship is no longer participating. The ethernet load savings in this scenario is also diminished if a player performs frequent actions. Both the less frequent position / course method and the course change only option require that ships positions are plotted, and that when a ship adds itself all other ships respond with their course and position information. I didn't believe these were desirable alternatives as they traded compute time for ethernet load. As suspected and shown in table 5.2, compute time is more important as display operations put a heavy load on the workstation.

# of Ships in game and SRS display	Display Time	Com. Time	Control Time	Total Time
1	181,263	55,014	31,457	267,734
2	316,824	50,936	28,184	395,944
3	212,344	53,496	25,984	291,824
4	378,151	64,690	27,774	470,615

Table 5.2 CPU utilization in microseconds.

5.2.2 Base Collision Packet

Base information handling is a special situation. Bases are semi permanent system objects. They are similar to a planet in that they are fixed within a solar system. However, they differ in that they are deployed by ships and may sustain damage or be destroyed. When ships enter a solar system they must know about all bases in it. When a base is destroyed everyone must be informed.

Three methods exist for base damage calculations. One is a server method where the server negotiates all base activities, monitors all solar systems and maintains base information. Another, is a method whereby base owners monitor all solar systems and determine base damage. Finally base owners can maintain information about bases and be informed by ships detecting a base / object collision.

The server method allows for robust base control and can be modified to make the bases aggressive, but requires a server machine which I wanted to avoid. Having owners monitor the bases and compute damage is also robust but requires excessive processing for the ship to monitor all solar systems and calculate all torpedo paths. The optimal solution is to have owners maintain the information, but have other ships determine collisions, thus distributing collision detection to all ships which are already doing it. Ships monitor torpedoes for base collisions so they may remove the

destroyed torpedo. They also monitor their own collisions with bases and other objects for damage assessment. This scenario requires a ship to transmit a notification packet to the base's owner for damage computation if it or a torpedo it is monitoring collides with the base. Also required is for the ship to test its phaser fire for hitting a base and report it to the owner. This test is the only extra action beyond notification.

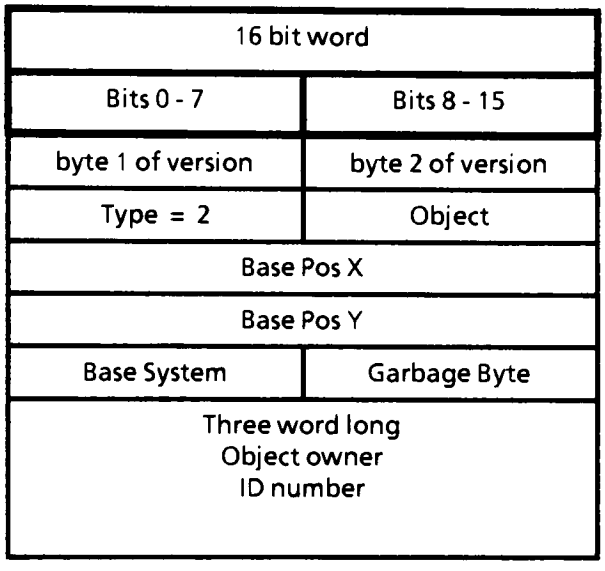


Figure 5.3 Base Collision Packet.

The Base Collision packet, type two, is shown in figure 5.3. It is sent immediately when a ship detects a collision between an object and a star base. It is sent specifically to the owner of the base, rather than being broadcast to the universe as other packets are. There are two draw backs to this method. The first is that if multiple ships see a collision the owner must detect that they are reporting the same incident and only asses it once. The other is that if a base is hit, but no one sees it, or the collision packets are lost, no damage will be assessed. A rule, that only one object type from a given owner may collide with a base in a second will prevent duplicate collision assessments. As torpedo fire reload cycles will be longer than one second, this rule will not eliminate legitimate sequential collisions of the same object from the same owner. To further reduce this problem, only torpedoes need be checked for multiple collisions. Bases are allowed to overlap stellar bodies and one another.

Two bases however may not occupy the exact same spot in space. Base deployment is blocked in this case. A ship colliding with a base or hitting one with phaser fire is responsible for notifying the base's owner of the event. Thus, only torpedoes may be multiply witnessed. The witnessing range for torpedoes is ship and torpedo dependent. A ship only tracks a torpedo if a chance exists that the torpedo will be displayed. This is dependent on the torpedoes range and the ships ability to bring that range into the short range scan display. Collisions not reported will be treated as base defenses eliminating the threat before impact.

5.2.3 Phaser Packet

The firing ship broadcasts a type three packet, figure 5.4, indicating the shot's starting position, its direction and range. Each ship in the specified system clips the shot to its short range display and displays it as necessary. It further decides if it was hit by the shot. If so then damage is assessed. Phaser shots are instantaneous and have a fixed range. So, it is not necessary to worry about where they go over time. They also are not blocked, but pass through objects to hit their target. I originally planned to allow blocking, expected cpu costs were to high to test every object for a hit and adjust the display to only hit the closest one. It was quicker to display the entire streak and only test the local ship, except the firing ship which checks for bases. Lost packets are considered intermittent weapon system failures (the sender reloads and tries again, potential receivers perceive no action).

16 bit word	
Bits 0 - 7	Bits 8 - 15
byte 1 of version	byte 2 of version
Type = 3	Race
Start Pos X	
Start Pos Y	
Direction	System

Figure 5.4 Phaser Packet.

5.2.4 Torpedo Packet

The torpedo packet is the same as a phaser packet except that it is of type four. Figure 5.5 graphically displays the packets layout. The firing ship broadcasts this packet to all other ships. Those in the specified system enter this information in their torpedo list. All ships are then responsible to track the torpedo with respect to their own position and display. They determine if the torpedo collides with a star base, another ship, a fixed body, or themselves. In all cases the torpedo is removed from the display and list when a collision is detected. That is all that happens if the object was a fixed body or another ship. The other ship is responsible for assessing its own damage. If it was a star base, the base owner is informed through a base collision packet as previously discussed. Finally, if it was with the local ship, damage is assessed as discussed in the following section. The only ethernet transmission associated with a torpedo is when it is fired. After that it is up to the individual ships to track and plot their own view of the torpedoes life. The torpedo will start ahead of the firing ship and travel at a speed greater than the fastest ship, thus it can never collide with the firing ship. As they are for phaser fire, Lost packets are considered intermittent weapon system failures (the sender reloads and tries again, potential receivers perceive no action).

16 bit word	
Bits 0 - 7	Bits 8 - 15
byte 1 of version	byte 2 of version
Type = 4	Race
Start Pos X	
Start Pos Y	
Direction	System

Figure 5.5 Torpedo Packet.

To prevent torpedoes which miss their targets and all other bodies in space from living forever, it is necessary to determine when they should be deleted from local lists. I considered three

possibilities. One was to ignore and remove from tracking all torpedoes not in the short range scan. Another was to let them run until they hit the outer bounds set up for the solar system. Yet another, was to assign a time limit to them after which they die. All have good and bad points. Only keeping those in the short range is efficient and saves time and space by eliminating many extraneous torpedoes. However, it prevents torpedoes fired off screen from entering and hitting a target. The next option of waiting until they hit the edge of space would keep a tremendous number of unnecessary torpedoes, potentially consuming all compute time to plot their positions. The final option of assigning a time limit, keeps some unnecessary torpedoes, but allows them to enter from off screen. The time limit can be seen as an effective range for the torpedoes. Therefore, I implemented this solution.

5.2.5 Intergalactic Message Packet

Intergalactic messages are type five packets. Figure 5.6 shows the packet layout. As with any other radio transmission, all ships in the universe will receive the transmission. The message will be displayed in the message subwindow of all receiving ships. If the message packet is lost, it is considered to be a radio disturbance in the intergalactic ether and the message must be manually retransmitted.

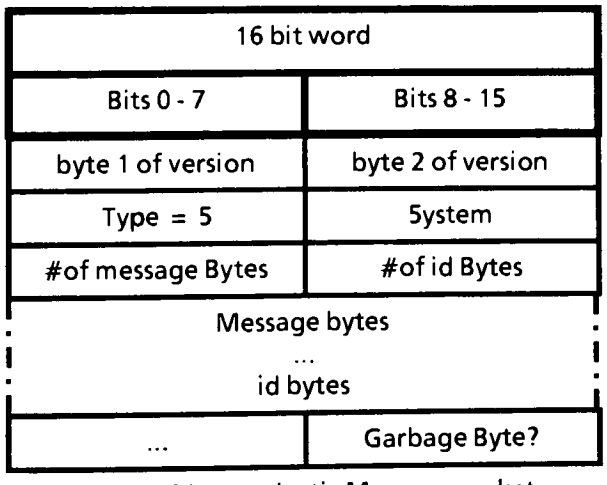


Figure 5.6 Intergalactic Message packet.
Max # message of Bytes = 128.
Max # of id bytes = 20.
Maximum size = 152 bytes.

5.3 Display

The local ship is displayed in the center of short and long range scans . The short range scan is a 4X view of the long range scan, clipped at 2X. To keep the local ship in the center of the display, the ship's new position is plotted at the beginning of each display pass. The fixed bodies are considered stationary objects in the local system and are thus shifted to match this change of position. All ships in the local system are then plotted for display with respect to the ship. Each torpedo's position is then calculated, plotted, and clipped for display with respect to the ship.

5.4 Damage

Total damage is computed, much as it was in the old game, by summing fixed and variable components of each instance with the current total damage. Table 5.7 shows the fixed base damage caused by collisions or energy blasts. Energy Blasts are the result of phaser fire or exploding star bases. The variable portion of collision damage is the relative velocity between the ship and object. For energy blasts, the variable component is based on the distance between the energy source and the ship. For star base explosions, this value is (64 - the ship distance from the base)2. For phasers, the value is (depth - distance)2, where depth is the phaser's range and distance is the ships distance from the energy source. Table 5.8 shows the depth value for each race. Shields remove (shield level)^2 x 25 points of total damage, before it is charged to the ship. After absorbing the damage, the shields are lowered one level. Possible shield level values are show in table 5.8 by race.

Table 5.8 also shows total maximum damage by race. When a ship's total damage reaches this level the ship is destroyed. For each damage instance, the damage received by a ship is randomly distributed among 1 - 3 of 14 possible damage areas (Phasers, Torpedoes, Attitude Control, Acceleration, Shields, Short Range Scan, Long Range Scan, Cluster Scan, Communications, Energy Systems, Hyper Drive, Velocity Display, Energy Display, Battle Computer). Table 5.9 shows how the damage level is escalated from none to severe for each of these systems. When damage is greater

Object / Energy Source	Base Value
Star	30
Comet	12
Small Planet	13
Medium Planet	14
Large Planet	15
Cluster	16
Asteroids	4
Any Ship	10
Romulan Torpedo	16
Klingon Torpedo	12
Terran Torpedo	12
Any Star Base	16
Exploding Star Base	0
Romulan Phaser	10
Klingon Phaser	14
Terran Phaser	10

Table 5.7 Object and Energy Blast fixed damage value.

Race	Maximum Damage	Shield Level	Phaser Depth
Terran	1200	0 - 3	30
Klingon	1200	0 - 3	35
Romulan	1400	0 - 4	25

Table 5.8 Race Maximum Damage, Shield Levels, and Phaser Range table.

0	< 40	< 200	> = 200
None	Slight	Moderate	Severe

Table 5.9 Damage Level / Point Mapping.

than slight the system will cease to function or function improperly. While slight each point of damage for the torpedo or phaser system is an extra millisecond for torpedo load or phaser energize time.

6.0 Important decisions

Throughout the development of trek several important decisions were made which shaped its outcome. Some of these decisions have already been discussed. Primarily these dealt with the communication system, packet layout, and base collision detection. Decisions which have not been discussed are the software layout into modular sections, object shape, and the bitmap displays.

6.1 Communication system

Section 5.2 describes the communications implementation for Trek. It also describes the decision to use a single well known socket for Trek, and the decision for distributed base collision detection versus a centralized service. However, the more global decisions to not have a game server and to use the Packet Exchange interface have not been discussed.

In choosing the communications system for Trek, I reviewed the games discussed in section 2.4, Summary of related work. The options presented were to use one of the players' machines as a server,as done in Tank, to use a collection of master and slave servers, as done in Blub, or to broadcast to a single socket, as done in Mazewar.

I was opposed to the server concept. It has several problems. For it to be a player's machine puts a heavy load on that workstation. The workstation not only drives the player's display, computations, and command input, but also drives the game for all other players. Further, if the player stops playing or shuts down his machine the service must move, and it may not be possible for it to retain the game's information. A game like Tank which only has two players and one display view can use this. The workstation demand is less and if a player leaves the game or stops his workstation the game is over. Trek is to large for this method. The load on the workstation is extensive and could not support

the additional load of a server. Further, if the server machine were to stop participating, data would be lost transferring to a new server.

If a server is used and it is not a player's machine, then it must be dedicated to the game or some other function which can be interrupted when the game is played. This is more than I wanted to ask of a site. Most workstation installations do not have the resources to supply a game server or a workstation which can drop its normal function to run a game.

Therefore, I opted for decentralized processing by all players over a server design. This does have its draw backs in that it has a WYSIWYG (What you see is what you get) effect. If everything goes right, this is not a problem and everyone sees the same thing. However, it is possible through dropped packets for players to see different things. For example, a player misses a packet which moves a ship out of a torpedo's path. He therefore sees a collision and eliminates the torpedo believing it to have hit the ship. Actually no such collision took place and the other ship assesses no damage. Further, it continues to monitor the torpedo. It may then witness the same torpedo continue on to hit the first player's ship. Of course, the first player has deleted the torpedo and perceives no collision, and assigns no damage.

The chances of such errors are very small and to the individual players there is no perceived problem. In fact, it is more aesthetic in that for each player what he saw is what happened. Neither realizes that the other did not sustain damage only that the torpedo disappeared when they saw it hit something as it should.

This decentralized method is used by Mazewar. Its implementation is through the XNS Pilot socket interface. Socket is a non-public interface which means that it is not guaranteed to exist or remain the same from one release of Pilot to the next. Reading the Pilot documentation I discovered Packet Exchange which sits on top of Socket. Packet Exchange is a public interface and guaranteed to

exist and supply the same functionality from one release to the next. I would like this game to be easily upgradeable from one software release to the next. Further, I hope to port it to other platforms. For these requirements to be met, it was best to implement this with Packet Exchange. While this merely sits on Socket providing me with functionality I do not use, it is guaranteed to exist in future releases even if Socket is eliminated.

6.2 Software Layout

Very early in the planning stages of Trek I envisioned it as three interrelated, but separate components. These components were communications, control, and display. I was able to design, develop and implement communications as such a separate piece. I also designed and implemented control and display. However, during implementation it became apparent that control was heavily linked to display. Further, control was really several pieces, player command entry, ship movement, and the data base link for display and communications. Player command entry came through the users window interface and was thus linked to display. Ship movement drove the display. What remained were the data bases which I tried to design and develop as individual entities. The data bases were easy to design for communications data input. However, links to the display and data consumption as well as inspection posed many problems. These problems are discussed further in the lessons learned section.

6.3 Object shape

For programming ease, all objects regardless of display shape are considered to be square by the game. The square is their bounding box. It is significantly easier to test for intersection (collision) of boxes than it is for circles or irregular shapes. The game uniformly uses boxes for this function.

Boxes are even used for phaser and base explosion preliminary tests. In this way, objects which might be hit by these energy weapons are quickly identified. These instances however have a second

level of testing for the possible objects. For explosions, it is range. The effect of a base explosion is circular. For phasers, it is range and actual intersection with a phaser streak. The box in both cases quickly identifies a small set of objects for the more time / cpu consuming test or eliminates the need for such a test.

6.4 Bitmap displays

There were many decisions involved in designing the bitmap displays. The first decision was to use constant bitmaps for all fixed display objects. These included stellar bodies, starbases, and torpedoes. This was an easy decision as none of these objects have a distinctive front. How to link the bitmap to the objects position in space however involved more thought, and how to draw the objects with a directional factor, ships and their shields, was still another issue.

6.4.1 Position selection

Body and base positions refer to the upper left corner of the object. This is convenient for drawing and collision detection. The size of the object is known and the object's box is easily determined from this corner position. Ship and torpedo positions are their centers. This means that their corner must be computed for drawing and collision detection.

The reason for the differences is that bases and bodies are fixed in space. They have an upper left corner position and dimensions which are necessary to draw their bitmap and determine collisions. This data never changes. If their centers were used to record their location, the position value would only be used to compute the upper left corner whenever it is needed. Therefore, that value is computed once and kept. The unnecessary center position is never referenced.

Ships and torpedoes however move, and have operations performed around them. To draw the ship's shields it is easier to move a set distance in all directions than to move more to the right and

bottom, and less to the left and top. Torpedoes are a single point in space, but they have a proximity fuse for 1 unit in all directions. Therefore it is desirable to keep the center position for these items. It is moved and then used to compute the object's display box, and the boxes that surrounded it.

6.4.2 Ship and Shield Bitmap Generation

Several options were available for bitmap generation. They ranged from rotating on the fly a 0° bitmap, or set of points from which the ships could be drawn, to displaying a static set of bitmaps. The static set was chosen, but then decisions on the sets granularity and how to create them were necessary.

6.4.2.1 Rotation on the fly

I initially considered the rotation option. I had hoped to display the actual ship angle rather than something close to it. However, the cpu cost was to high to rotate n ships 4 × 4 and 16 × 16 bit patterns. I considered reducing this cost by identifying structures within the ships bit patterns in hopes of rotating only a few points and then applying a set of commands to them to draw the ships. The number of points necessary was still high and no significant savings could be realized so I opted for the fixed set of bitmaps.

6.4.2.2 Granularity of static bitmaps

I built a set of tools to allow me to choose the granularity and test it. By setting a global constant to the desired angle. The appropriate data structures would be generated and a static set of 0° bitmaps would be rotated and placed in the structures for both ship and shields. I tried several granularity values. What I found was that it was aesthetically important that 0°, 90°, 180°, and 270° be accurate. They drew the nicest and it seemed that they should be accurate no matter what the other angles looked like. I also discovered that below 20° the size of the data structures necessary to hold the bitmaps became to large for a Mesa module. Therefore, 30° was chosen as the optimal

granularity. It was the smallest value greater than 20° which met my aesthetic requirement of being a factor of 90°

6.4.2.3 Bitmap Creation

My first software releases continued to use the tool set mentioned above. However, this required the player to wait for bitmap generation, rotation to all 12 positions, before beginning the game. Therefore, Using the debugger I stopped the game during play and extracted the rotated bitmaps. I then placed these in the source code as initial values for the data structures and removed the bitmap generation tools. These bitmaps were acceptable, but the long range scan appearance at angles between 0°, 90°, 180°, and 270° were not very appealing. Therefore, to continue enhancing the aesthetic value of the bitmaps, I used a bitmap editor to redraw the ships for the odd angles 30°, 60°, 120°, 150°, 210°, 240°, 300°, 330° This however has resulted in their not being drawn at exactly the 30° increment, but rather somewhere above or below 45° increments as appropriate for enhanced appearance. I am continuing to enhance the aesthetic quality of the bitmaps.

7.0 Conclusion

This thesis was significantly more than I anticipated. Throughout its development I resolved my open issues and learned about Tajo, Pilot, and the benefits of planning before implementation. I have also opened some presently unresolved issues.

7.1 Resolved issues

7.1.1 Will the software run fast enough to give the user a responsive interface, and how frequently should status packets be sent (is 1 sec. sufficient?)?

The software running at a one second game interval is acceptable. However, some people have expressed that they feel the game's display should update a little faster. Optimally the status packet transmission and display update rates are equal; each update is displayed. Network loading is not a factor and packet frequency can meet or exceed the display rate. The display speed is an issue. The current cpu load is approximately 75%. I have tried a display rate of .75 seconds. This does quicken the game, but loads the cpu at 100%. Anything beyond casual sailing around puts the game in a catchup mode. Therefore, I have not increased the display rate. To do so requires better optimization of the software, or more cpu power than is presently available on the Xerox workstations.

7.1.2 Will the ethernet load be excessive?

I was correct about cpu time being more important than net load. A game with several players has almost no impact on the local ethernet.

7.1.3 How much other work may run in parallel with Trek?

The cpu load is currently about 75%. This is not excessive, but does occupy the cpu with the game. While other activities may occur simultaneously, it is obvious that the cpu is thrashing between them, and the real time features of Trek suffer.

7.1.4 Will Packet Exchange really do what I think it does?

I was correct about Packet Exchange. With only a couple minor hitches, due to incorrect documentation, I was able to utilize packet exchange as planned. This was the easiest part of the project as it was planned, implemented, and tested almost without flaw.

7.2 Lessons learned

As already mentioned, this project was more than I anticipated. I learned a great deal while working on it. Some of the specific things learned are the Tajo window system, Pilot's Packet Exchange communications system, and that planning before implementation does work.

7.2.1 Tajo window system

My initial results with the window system were quite good. I already understood the common window and subwindow commands. What I needed to learn was how to create my own subwindow types and how to link them with other windows in the desired layout. There was no common facility for creating the long and short range scan graphical displays or for mixing my other graphical displays with the standard subwindows. However, I was able to use the basic windowing tools to create my own subwindows and to merge them with the standard types in ways not commonly done.

7.2.1.1 Graphic Subwindow

The graphic subwindow I required for all of my windows uses the Tajo Display interface to manipulate them. To create these subwindows I used ToolWindow's CreateSubwindow command. I supplied it with the size and position for the subwindow as well as the subwindows call back display procedure.

7.2.1.1.1 Display call backs

The call back display procedure is called by the windowing system whenever some portion of the window becomes visible. The newly visible areas of the window are supplied to this procedure as boxes which need to be painted. The call back procedure paints these areas according to the subwindow's needs.

The subwindows in the Status, Command, Message, and Galactic map windows simply redraw their contents. All of them except the Galactic map are small and contain only a few objects. This makes checking for intersection with the invalid window areas more time consuming than it is worth. The Galactic map however could save time by doing such checking, but I did not implement it in such a way as to make this easy, and it is left as a new open issue.

The Long and Short Range scan displays do use the invalid box information and only display the necessary ships, torpedoes, bases, and bodies for these areas. This is where I had one of the hardest times developing the code. Because objects overlap on the display (collisions), I exclusive-or the bits to the display. A subsequent exclusive-or will erase the object and leave any object it overlapped properly drawn. When repainting a box I only want to draw the part of the object which is missing. The way the bitmap displays work this cannot easily be done. The entire object is easiest and quickest to draw. This would erase the currently visible portion of the object and display the missing part. Therefore, I drew all of the objects intersecting the box; erasing their visible parts. Then I erased the

box and drew the objects again so they were completely visible. This action took a very long time and needed to be protected against a change in game or display state during the entire operation. It developed a problem with the notifier for the Tajo environment in that it could change my state and attempts to lock out the notifier developed deadlock situations. This is when I accidentally learned that none of this was necessary.

The ship displayed in the command window is displayed in a manor similar to the one in the Short scan except that it does not overlap anything. To prevent the problems experienced by the short range scan, instead of exclusive-oring the bits they were to be painted without any test. However, I created this code by copying the Short Range scan display command and failed to edit the bit function. The display worked as it should even when only part of the ship was obscured. One day when reviewing the code, I came across the error and realized that the display which worked, shouldn't. Further investigation proved that though undocumented when the display procedure is called the only area of the display it can manipulate is the supplied box. This was a very expensive lesson to learn which wasted a tremendous amount of time.

7.2.1.2 Subwindow layout

The standard subwindow layout for Tajo is for them to be stacked one on top of the other taking the full width of the parent window. For Trek, I needed to have windows split left and right, take only a corner, or both. This too was not documented, but with some experimentation I was able to learn how to implement my windows. To put a subwindow on the right side of its parent was easy. The subwindow command took the windows position and dimensions. So it was only necessary to specify a position appropriately within a parent window. The command however assumed the width to extend to the parent window's right edge. Therefore, subwindows on the left would overlap subwindows on the right. The same problem existed when taking only a corner and leaving the rest of the window to another subwindow. In order to overlap subwindows, the top subwindow needs to

be specified first. One would assume the bottom subwindow first, as it is for windows. However, that results in the desired bottom subwindow being placed opaquely on top obscuring the subwindows to its right.

7.2.2 Pilot communications

At the start of the project I was unfamiliar with the communications interfaces supplied by Pilot. Therefore, as mentioned in the previous section I reviewed similar applications for my communications. After selecting Packet Exchange, I built some prototype test code and experimented with it before implementing Trek through it. During these tests I learned that there was an error in the Packet Exchange documentation, and that one method of status transmission I had considered would not work.

The documentation error that I discovered was that the timeout for no reply was documented as 0 meaning no wait. Instead 0 means infinite wait. This became very apparent when tested, the test never completed.

The implementation plan that failed was a hope on my part that the pointer supplied to packet exchange as the packet's data would be used in timeout retransmission. It is not . The data pointed to is copied by Packet Exchange and that copy is used for retransmission. My plan had been to supply Packet Exchange with the location of my status packet and set the packet timeout for no reply to the games status transmission rate. As no replies are ever sent, I hoped this would allow the game to alter the status packet while packet exchange handled periodic transmission at the lowest possible level. I learned otherwise. Though I could have gone to the necessary level to implement this functionality, as mentioned earlier, I desired to keep this game as portable as possible. Thus, I implemented the appropriate loop for retransmission on top of packet exchange.

7.2.3 Plan then implement

Just before beginning this project I attended a structured programming class. I was not convinced by the class that planning before implementation really worked. All of the test cases, examples, and problems were well defined. My impression was if you know everything about the subject, can document it, and have your document reviewed by someone, then you can quickly turn your document into code and have a perfect implementation.

In reality, as in this development, you do not know everything there is to know about the subject. Therefore, I planned to use Trek as a test case for myself. I planned to sit down and plan each interface before implementation. I did this, but as predicted implementation did not go perfectly smoothly. I initially considered this evidence against the methodology. However, when I began to document it I realized my problem was that I did not plan well enough.

I thoroughly planned the communications part of this project . This worked extremely well. The only changes made to the packet layouts were addition of ship shield information. This allowed the display of ship shields which was not originally planned. The first coding I did was a feasibility program using static data to see that my understanding of Packet Exchange was correct . Once I selected how to startup and shut down the communication interfaces for the game, I never changed any part of the communications code.

The data bases which link the communications and display packages for this game were also planned. For each of the data bases, I planned all the different routines that would be necessary (start up, shut down, data update, display access, collision detection, watch dog, ...). I implemented these and tied them to the communications and they worked perfectly.

Next I attempted to plan the game's display. I even built a feasibility model to test it. With some effort I constructed the windows I needed the way I wanted them to look. I was able to display sample bitmaps, and felt comfortable that I could go ahead and implement. This is where I erred.

The display was not planned to completion. Specifically I had not experimented with ship movement. It all seemed so easy and straight forward. I had proven I could draw the window and display / erase ships in it. The Display interface for Mesa had a Shift command and everything seemed to be in place. Then I implemented the shift and discovered complications when the window shifted while refreshing an area. At this time, I should have gone back to planning, but instead I continued implementing. This led to experimentation and modification of the whole program which resulted in the problem migrating into the data bases.

The display documentation for a shift which is how I move the ship through space by erasing it then shifting the visible portions, says to perform the shift , update your data structure then invalidate the areas shifted from, then validate the subwindow to call the subwindow call back procedure. Shift does not call the display procedure so as to avoid confusion [9]. Based on this I attempted to circumvent my call back problems by not invalidating and validating the window. I tested this and it seemed to work. However in application it only works for about 5 minutes and then the cpu locks up at 100%. The problem, after much searching without finding anything, is that the invalidate / validate calls do something more than call back the display procedure for the window, and failure to perform these activities causes cpu lockup over time.

Through all of this I learned that the plan before implementation method needs to be tempered with some feasibility models which thoroughly test understanding of the problem to be solved. As shown above, in the areas where the plan was sound and feasibility testing confirmed it, the plan did implement well. This can be seen in the communications and data base cases. However, if the plan is

defective and the feasibility test does not catch it, the implementation fails miserably. This is evidenced by the display case.

7.3 Unresolved issues

Throughout the course of this project several issues have come up which are still unresolved. They are code optimization, phaser display, input focus, and portability.

7.3.1 Optimization

Code optimization is a way to enhance the game possibly increasing its display update rate or the amount or work done in parallel with the game. Areas available for update are selection of the solar system warped to, the galactic map display, and any computations especially those using real or long numbers.

7.3.1.1 Warp jump system selection

Currently warp jump system selection is computed by following a vector from the current system in the ships velocity direction to see if it intersects a system in the galactic map display. A more efficient method, not implemented only because of time constraints, is to create a large nested select statement which has for each current system the system warped to for a given angle or range of angles.

7.3.1.2 Galactic map display

The galactic map display needs to be reviewed. It is possible that setting it up as a binary tree like the solar system bodies would help, but I'm not sure. It would certainly help for partial displays, but the galactic map is usually completely redrawn if it is redrawn at all. Therefore, I did not invest time in optimization as there was little to gain. It is however still a potential savings.

7.3.1.3 Computations

Without hardware upgrades only available to the 6085, the Xerox workstations are not well equipped for mathematical operations. Therefore, any computations especially real or long operations should be avoided. During implementation I tried to keep this in mind and avoid the math where possible. At this time however, it is evident that some of the computations can be simplified or eliminated through the use of special constant values. An example of this is phaser hit testing. If an object is in the vector path of a phaser, it is necessary to test the range. This involves taking the square root of the x distance squared plus the y distance squared. The optimized solution is to compute $x^2 + y^2$ and compare it to $range^2$, thereby eliminating the square root function. $Range^2$ is a constant computed at compile time off range which is a constant defined for each phaser type. Many situations such as this exist and I am optimizing them as I find them.

7.3.2 Phaser Fire display

When a phaser is fired, lines are drawn to represent each of its streaks. To erase the lines I invert a box which encompasses the streaks,redraw them, and then reinvert the box back to the normal background. Unfortunately, this process returns all bits in the streak to the background color. This leaves bits which should be inverted for an object hit by the phaser uninverted. When the ship is erased for destruction or movement, it exclusive ors its bits to erase itself, but actually draws the missing bits back on the display and leaves them behind.

I have attempted to rectify this problem. Initially, I tried to draw each bit myself by inverting it then inverting it back to erase it. This, however, from a top level, takes considerably longer than to have the system do it, and was unacceptably slow. Therefore, I returned to the original methods and let the phasers carve pieces of the object off and leave them behind. An accepted fix might be to

detect the objects which are hit and redraw them. This again however is time consuming, and with the cpu already at 75%, it is left as unresolved.

7.3.3 Input Focus

A work around for the phaser problem was a screen refresh command. However, this is linked to another problem, input focus. Text typed to any trek window should go to the outgoing message string field. I have been unable to resolve how to get the input focus for my windows. This causes Trek to only see the mouse, not the keyboard and mouse. This has been an issue throughout development. I expect that through trial and error I will eventually resolve it. As far as I can tell, I am doing what is necessary, but this part of Tajo is not well documented.

7.3.4 Portability

Earlier, portability was discussed with regard to the communications interface decision. At this time I still hope to port this game to the Sun environment, but have no word as to availability of the Tajo and Pilot systems on it. I believe the increased cpu power of the Sun would support a faster game interval and allow more parallel activities. I am also interested in Sun's open look interface and interface building tools. The future of this game is in its portability.

7.4 Conclusion

Overall, I feel that this project was very challenging. I look forward to my game's distribution and use. I have learned many things through the experience both in Mesa programming for the Xerox workstations and in programming in general.The lesson learned regarding structured programming is specifically good and practical. I had always questioned the theory because its presentation and my experiments with it were canned. Thus, the planning was easy and the implementation straight forward. This program provided me with an example where design was not easy nor the implementation straight forward. I failed to properly verify my plan and paid the price in

implementation time. In review, the error is obvious and thus has more meaning than all of the successful class room problems.

Appendix A

Definitions

CSMA - Carrier Sense Multi-Access.

Datagram - Data packet sent without concern for order. It expects other devices to take care of order and duplication.

Deadlock - Monitor condition where all processes are waiting for a condition which will not or can not occur.

Ethernet - CSMA communications network.

Experimental Ethernet - 3 megabit PUP network developed at PARC to demonstrate ethernet feasibility.

In line procedure - An in line procedure has the call to it replaced with the procedures text modified to use the supplied local parameters. It trades increased object code size for reduced run time through elimination of procedure call overhead.

Mesa - A modular programming language from Xerox [8].

Monitor - A collection of procedures for which it is guaranteed that only one will be executing at any time.

Packet Exchange - Pilot communications interface to be used by Trek [10].

PARC - Palo Alto Research Center.

Pilot - Xerox Workstation's Operating System designed to support Mesa [10].

PUP - PARC Universal Packet.

Socket - Handle through which host software communicates.

Tajo - Xerox Workstation's Development Environment User Window Interface.

TIP - Terminal Interface Program used to customize Tajo window input interfaces [9].

User.cm - General command file used by XDE tools to set initial parameters [11].

Viewpoint - Xerox office automation environment.

VP - Viewpoint.

Well-Known Socket Number - Socket numbers statically assigned to specific services and functions so that they may be known to all software applications [10].

WYSIWYG - What you see is what you get.

XDE - Xerox Development Environment [11].

XNS - Xerox Network System - a set of communication protocols from Xerox.

Appendix B

Source Architecture

B.0 Overview

The Trek source code is comprised of 7 definitions modules, implemented by 19 program and monitor modules. It is constructed from 8567 lines of code. Definitions modules supply a blueprint for how parts of a system fit together[8]. Program and monitor modules provide actual data and executable code to perform the actions defined by the definitions modules[8]. This section provides a graphical view of the software's procedural flow, and describes the source code by the definitions modules and programs / monitors which implement the them.

B.1 Software procedural flow

The following diagrams graphically display procedural flow within the Trek Game.

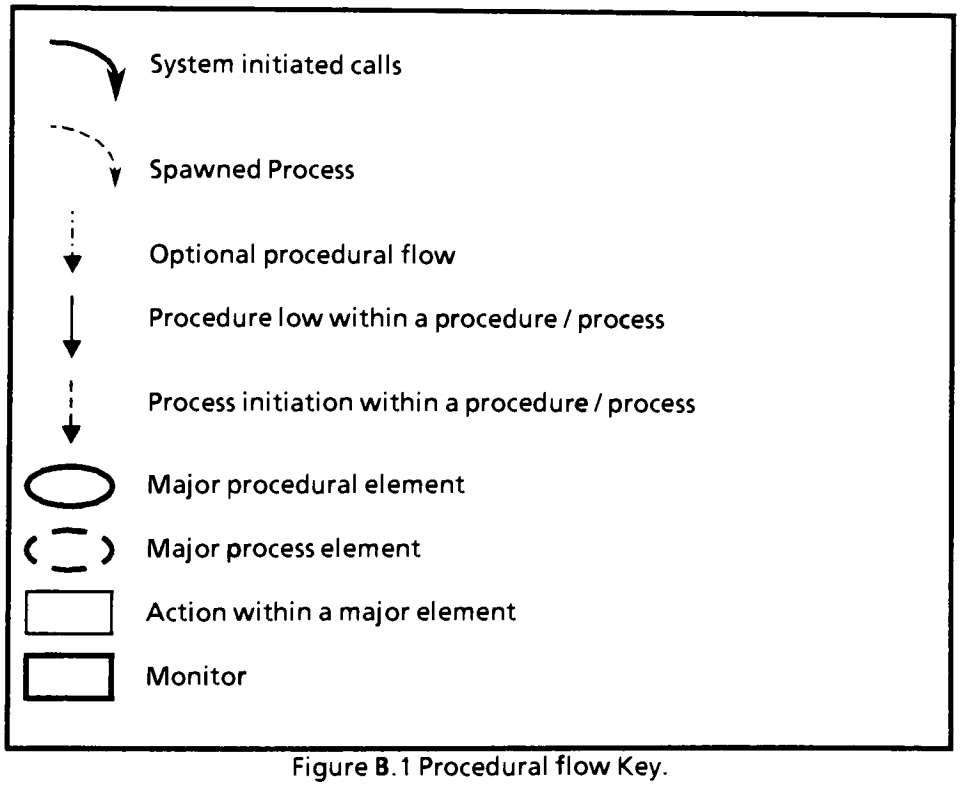
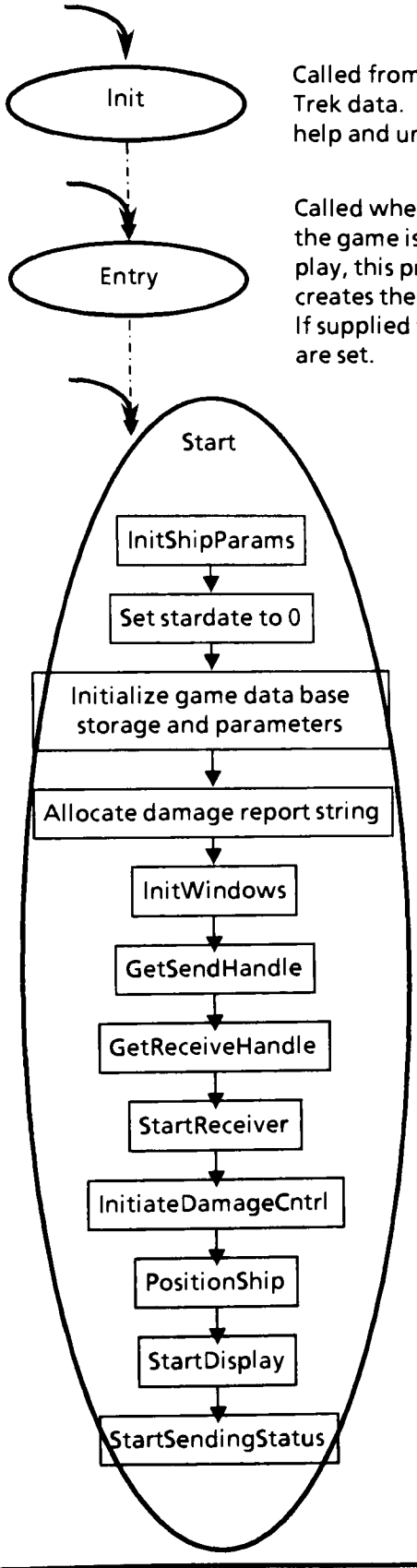


Figure B.1 Procedural flow Key.

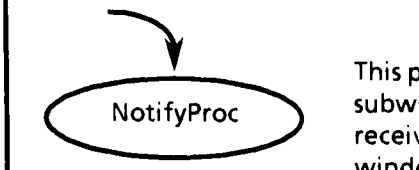


Called from the executive this procedure initializes the fixed Trek data. Further, it registers the trek command and trek's help and unload facilities with the executive.

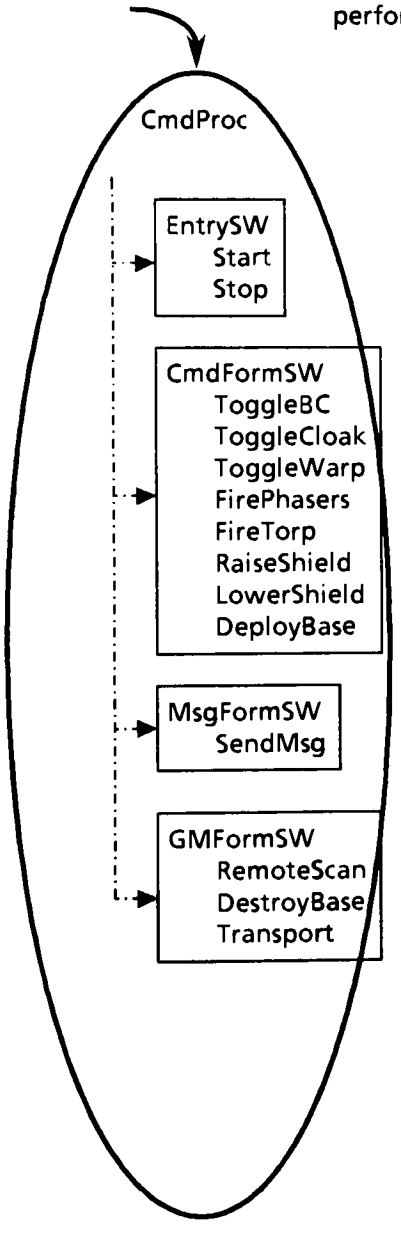
Called when the trek command is issued in the executive. If the game is in operation its windows are activated. If not in play, this procedure allocates trek's dynamic storage and creates the game entry window for initial parameter setting. If supplied through User.cm, the games initial parameters are set.

Called from entry if the User.cm has autostart set. Otherwise called by CmdProc when the start command is issued. This procedure initializes the local ships data based on its race. It initializes the stardate counter, the game's data bases, and the damage report storage. It then initializes the game windows and sets up the game's communications send and receive handles. At this time, the game is ready to play, but not yet started. To start the game, the receiver is started first. This gives the game information to work with. Damage control is started while the initial data is gathered. Using the collected information, the ship is positioned out of other ships torpedo range if possible in 10 random position attempts. With the ships position determined, the display and game control are started. Now that the ship can see its surroundings the local ship begins broadcasting its status packets, announcing itself to the rest of the players.

Figure B.2 Trek Initialization flow.

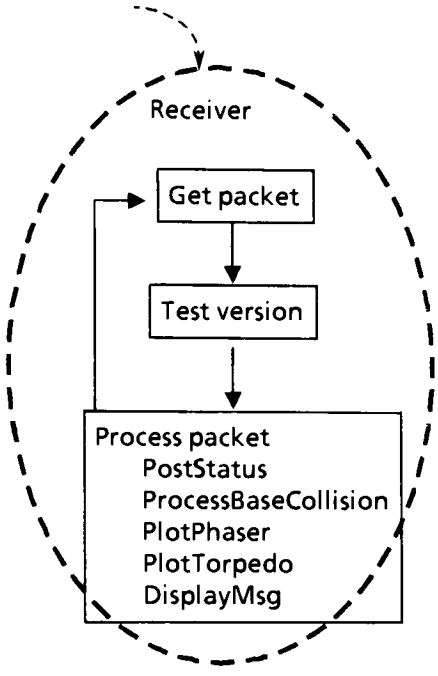


This procedure is attached to the long and short range scan subwindows and the graphical command subwindow. It receives game input from mouse actions in the game windows. Based on the window, location in the window, and action performed, this procedure determines and performs the requested user action.

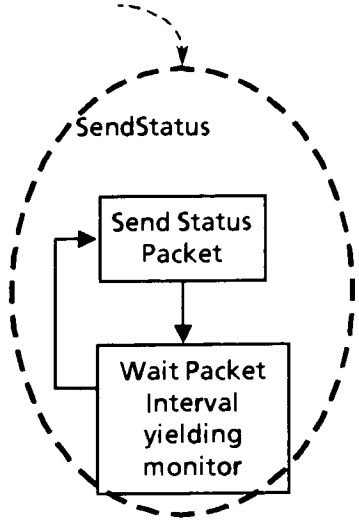


This procedure is attached to window command and boolean items. It is called by all window commands in the game. Based on the command issued, this procedure performs the requested user action, adjusting the local ships data and display to reflect the action. FirePhasers, FireTorp, and SendMsg each generate their respective data packet and broadcast it to all ships. For phaser and torpedo fire, see figure B.7.

Figure B.3. Trek Command flow.

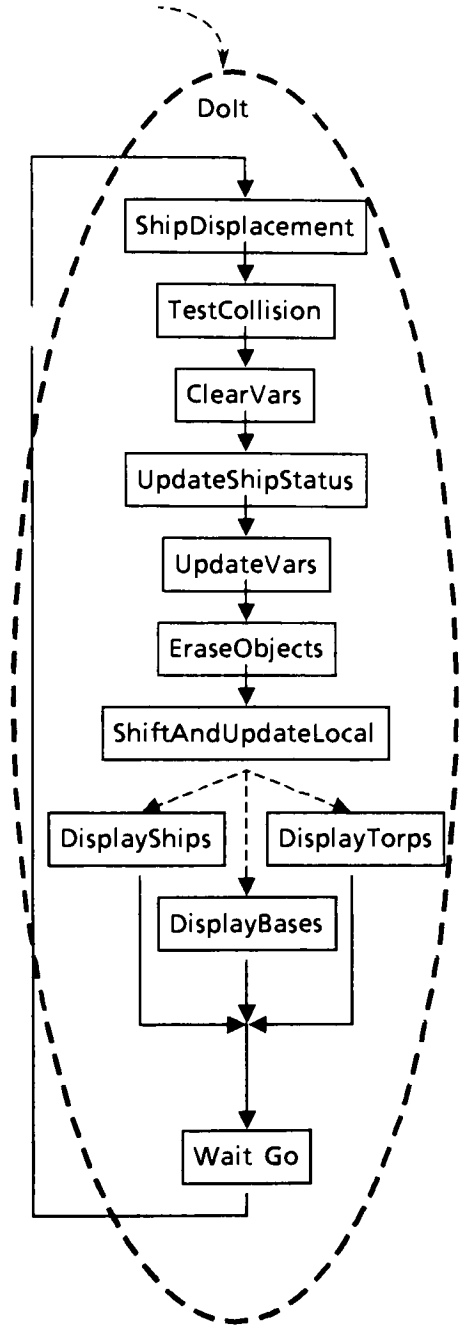


This process is spawned at game start up by StartReceiever. It is paused and restarted for when the local system changes due to a base transport or warp jump. The process waits for an incoming Trek game packet. If the packet is the current game version, it is processed based on type. Processing involves recording the data in one of the games data bases for future display and collision testing, or immediate display and collision testing. After the packet is processed, the process returns to waiting for the next packet.



This process is spawned by StartSendingStatus at game start up. The process iterates once every packet interval, sending the status packet and then sleeping until the next iteration. It is monitored to protect against simultaneous execution with UpdateShipStatus which is called from Dolt. This prevents a partially updated packet from being sent.

Figure B.4 Trek Receiver and SendStatus process flow.



DoIt is iterated every game display interval. It moves the local ship updating its local status information. It then tests the local ship for any collisions during the move. The ships status displays are cleared for any changes, its status record is updated from the more accurate local status information, and its status displays are repainted as necessary. With the local data updated, the scan displays are now cleared of any movable objects, The local ship and its shields are erased and the scan is shifted opposite the local ships movement to simulate that movement. Following the shift, the newly visible areas are automatically painted with the local ship and its shields. The movable items are then painted in parallel with any base changes which occurred during the last time interval. The movable items are panted in parallel so that any which are hung up waiting for a data base update will not block another which is ready to run.

The WakeUp process marks the individual display time units signaling DoIt to begin each control display cycle.

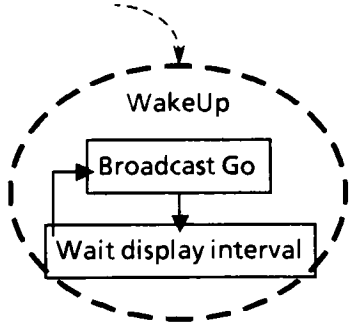
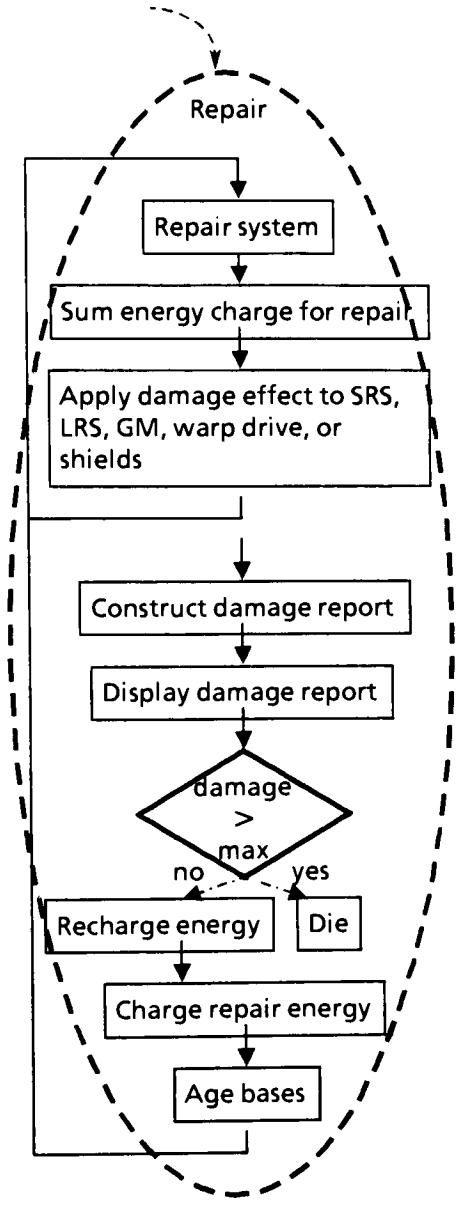


Figure B.5 Trek local ship control and periodic display flow.



Repair cycles once every display interval plus the time of repair. This makes a heavily damaged ship repair slightly slower than a ship with only a few damaged systems.

It first cycles through all systems repairing those that are damaged. The amount of the repair is typically 1 unit. However when the ship is docked the repair adjustment is increased to 4 units.

Energy charge for repairs is summed and charged after the ship is repaired. This is more efficient than multiple test and set operations. Also, it provides for an easier and cleaner exit if the ship dies from damage or lack of energy. The energy charge is typically 1 unit, but drops to 0 when the ship is docked and using the bases energy supply for repairs.

After damage is assessed to a system it is checked for a transition which affects the current game display. If one is detected the appropriate display modification is made.

When all systems have been processed, the damage report is generated and its display refreshed.

If the total ship damage is greater than the ships maximum limit, the ship is destroyed. Otherwise, its energy is recharged at the ships standard rate (unless the energy system is damaged). Then the energy charge for repairs is assessed.

The ships bases are then aged, increasing their energy 1 unit for normal bases and 4 units for bases near resources.

Figure B.6 Trek local ship Damage control.

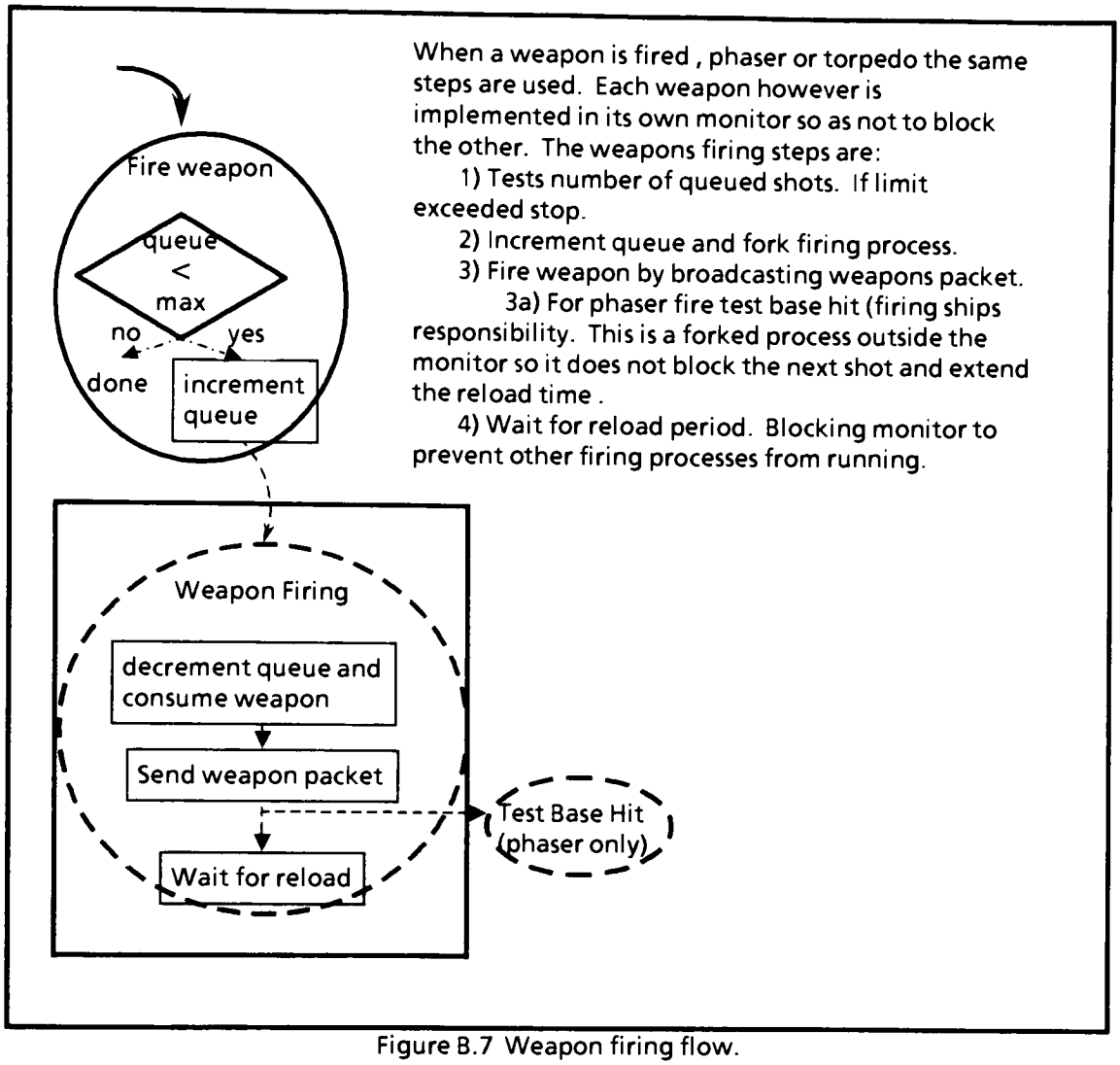


Figure B.7 Weapon firing flow.

B.2 TrekDefs.mesa

B.2.1 Static data

The primary function of this module is to define the global constants for the game / control part of Trek. It contains static information for the ships, bases, torpedoes, and phasers specified by race. It also contains the game information defining views, solar system size, and key ranges in game units.

B.2.2 Significant data structures

This module contains the record data structures for the ships game parameters. There are two such records. One record is the backing data structure for data supplied or displayed through the window interface. The other record is the ships local information. This record is the source for the ship's status record except that it has more data, which is more accurate than what is transmitted.

B.2.3 Significant global variables

TrekDefs supplies only one non-constant. It is the game's time variable, stardate. The actual data for this variable is allocated in and exported from TrekDisplayImplA. It is provided globally to the game so that any process can accurately find game time when needed.

B.3 TrekCom.mesa

B.3.1 Static data

This interface Supplies constants which are specific to communications. This information includes packet size, maximum message and identification lengths, version and socket identifiers, and the packet transmission interval values.

It also defines one in line procedure which constructs and sends a base collision packet. This procedure is in line because it is not complex enough to rate a procedure call. However the text of the call is not necessary in the calling procedures. It is more eloquent to enter SendBaseCollision [*information*] than to lay the information correctly in the packet and call the packet sending procedure. Therefore it is done in line, the parameters are automatically mapped to there packet locations and the packet send command is issued on it.

B.3.2 Significant data structures

One of TrekCom's primary functions is to define all of the communication packets listed in chapter five. They are defined here as machine dependent records for use in the communications implementation. The packets are machine dependent so that all workstations and all compilations yield the same data structure layout. This not only lets the local game's procedures interact through this interface, but guarantees a common interface for all workstations to share game information. The communications send procedures use them to allocate storage and assemble packets with the appropriate data for transmission. The communications receive procedures use them to decode packets which are otherwise seen as incoming byte streams. A retyping of the byte stream to the appropriate packet type yields easy access to its data elements.

B.3.3 Procedure Interface

The other function TrekCom performs is to define the procedural interface between the rest of the Trek game and its communication package. TrekComImpl implements the procedures defined by TrekCom. This function allows TrekCom and TrekComImpl to be independent of the Trek game. TrekCom defines procedures to get the handles for both sending and receiving. It also provides procedures to start automated sending and receiving of status packets. It provides the ability to send any packet desired. Further, it provides utility functions to update the status packet while blocking status transmission so as not to send confused data. Finally, it provides a means to halt a specified automated function, sending or receiving, to possibly be restarted later, or to be released which is the last function it provides.

B.3.4 TrekComImpl

This is the implementation module for TrekCom. It supplies the actual data and executable code that provides the functionality defined by TrekCom. The Trek game modules are only concerned with

two blind items which it requests and receives from this module, the send and receive communication handles. When the start up procedures are called the procedures in this module allocate the necessary variables and initiate processes to do the work returning these pointers. The module functions autonomously oblivious of the game. When the game wants to change something or perform a special function such as halting or stopping some communications feature, it calls a procedure which modifies the data in the module. The module handles this modification again without concern to the game. The key elements of this module, the receiver, status transmitter, and status update procedure, are graphically displayed in figure B.4.

B.4 TrekOps.mesa

TrekOps has only one function. It does not have any static data nor does it define an interface for game procedures. TrekOps only function is to define the global Trek game data structure. This structure is composed of game information and pointers to ship, communication, and display information. From this data record, all information important to the game can be found. It is passed as a pointer to most game procedures. The data record is defined in this special interface to avoid conflicts during compilation. The proper place might be thought to be TrekDefs. However, the data record references information from TrekCom which references TrekDefs in defining its packet data structures. Therefore, TrekDefs must be compiled before TrekCom, but this record requires TrekCom to be compiled before TrekDefs. To resolve the conflict, TrekOps was created as a top level referencing the supporting structures defined in TrekDefs and TrekCom.

B.5 TrekList.mesa

B.5.1 Significant data structures

One of the key functions for the TrekList interface is to define the data base record structures for Trek. This is in support of all the game's data bases (body, collision, display, player, remote scan, ship,

starbase, and torpedo). Their data structures are defined here for use in the implementation modules which write to and extract data from them.

B.5.2 Significant global variables and Procedures

The other function for this module is to define procedural interfaces between the eight data bases and the rest of the Trek game. All of the data bases except TrekBodiesList provide a similar interface. Each is a Monitor which has an initialization procedure, a cleanup procedure, and then depending on the data base requirements the interface provides data update and inspection procedures. These procedures will be discussed with the individual modules which implement them.

The initialization for all of these modules preallocates all expected storage for the data base. They use a heap based on a fixed record allocating all the available records from one or two pages of virtual memory. Should this be insufficient the data base is capable of allocating additional pages of storage on the fly. I attempt to avoid this as it is time consuming.

As mentioned earlier, TrekBodiesList is different from the rest of the data bases. The data base for the bodies does not change it is allocated as a variable being initialized so that pointers may be used, but it is otherwise a constant. The interface to it only provides read access to determine visibility of a body and proximity or collision with it.

B.5.3 TrekBaseList.mesa

Starbases being semipermanent items, this was the most difficult data base to implement. The module contains two data lists to access one data base. The first is the starbase data list. It is a binary tree sorted on the x coordinate of the starbase's upper left corner. The second list is a starbase display data list of starbases which need to be displayed during the next display update. It is implemented as

a linear list linked through the sorted list of starbases in the local system. The display update will be discussed in TrekDisplayImpl.

An update function is provided by this interface. This is necessary as starbases may sustain damage, be destroyed, or be captured and changed to look like a starbase of its new owner. Updates affecting a starbase's appearance cause the starbase record's pointer to be pushed onto the starbase display list. This list is painted to the display during the next display update. The bases are not updated immediately to avoid a conflict with the displays shift also discussed in TrekDisplayImpl.

When initialized, this data base starts up its own watchdog. The watchdog deletes any record not updated in three transmission intervals. If an expired starbase is visible in a display, it is linked into the starbase display list for deletion.

B.5.4 TrekBodiesList.mesa

This is a static list which provides to all games the same galaxy, with the same planetary layout in each solar system. The galaxy is implemented as an array indexed by solar system. The solar system body lists are implemented by flat binary trees sorted on the x coordinate of the upper left hand corner of the object. The root of each tree is a star whose left corner is at the center of the solar system. This list is scanned by the call back procedures for the long, short, and remote scan windows for display purposes. It is also inspected for collisions by the local ship and torpedoes. Further, base deployment inspects it for base proximity in determining the resource status of the base upon deployment.

B.5.5 TrekCollisionList.mesa

This is a linear linked list of torpedoes which have hit a base belonging to the local ship. As many ships may witness and report a torpedo hitting a base, this list insures that only one instance of the

collision is charged against a base. Incoming base collision packets scan this list for a match. If no match is found, a record is created and damage assessed to the base. If a match is found and the same base is being reported as hit within one packet interval, then the collision is assumed to be a repeat and ignored. If the time between collisions is more than one interval, then the record is updated to the new time and the base is assessed appropriate damage.

When initialized, this list starts up its own watchdog which deletes any record not updated in three transmission intervals. The reason for leaving the records in the list for more than one interval is that it is likely that the same person will be shooting at the same base and hit it multiple times. Therefore, time can be saved by finding a match rather than searching the whole list to determine this is a new collision. Along the same reasoning, when it is determined that a new record is necessary, that record is pushed on the top of the list as it is the most probable next item to be searched for.

B.5.6 TrekDisplayList.mesa

This list maintains the movable objects currently in the long and short range scan display subwindows. It is implemented by linked records containing the objects display handle, and screen coordinates for each subwindow. Movable objects are those which are typically in motion, ships and torpedoes. This list only contains display information for movable objects that are presently in the long or short range scans.

The list is created and emptied every second. When a movable object is displayed it is written into this list. Before the display is shifted to represent movement of the local ship, all objects in the list are deleted from the window and the list is emptied.

The list is also used to refresh the display, either by the refresh command or when the window is made visible after being obscured. This eliminates the need to search the larger ship and torpedo lists which include objects known not to be in the displayed scan areas.

B.5.7 TrekPlayerList.mesa

This is a simple linear linked list of players in the game. Every incoming status packet has its sender checked against this list. The counts for players in the game and local system can then be updated appropriately and displayed by race.

The list has its own watchdog procedure. It is started and stopped with data base initialization and termination. The watchdog removes any player not seen in three packet transmission intervals.

B.5.8 TrekRemoteScanList.mesa

The remote scan only displays bodies, bases, and ships, not torpedoes. Bodies are displayed when the window is brought up or refreshed. The scanning base does not move . Therefore, no display shifting will be done and objects outside the starbase's view can be ignored. The only objects which need to be recorded are the ships and bases which are in the long range scan view surrounding the starbase. This information is received from status packets.

The data base is implemented through two data lists. The first is a linear linked list of ships. The second is a similarly linked list of starbases. This starbase list is not a binary tree like the other starbase list as it is small, only containing visible bases. Also, it will only be searched for display. Every item in the list is in the display. Therefore, the time spent to keep the list sorted can not be recovered by reduced access time searching the list.

Update procedures are supplied for both data lists to be called by the status packet receiver. New objects and changed objects will immediately update the display. Once an object is displayed, only changes to it or screen refresh will result in future display of the object.

A watchdog process is supplied at initialization of the data base. Any object not updated in three status packet intervals will be deleted.

B.5.9 TrekShipList.mesa

This is a bidirectional circularly linked list. It is searched forward by update and in reverse by display. It is also searched during torpedo and local ship movement for collisions. This search is done in the forward direction. However, it is done completely within the monitor so direction is not important.

The intent of the bidirectional search is to increase data update time by having the incoming packet be the next data record, and to display the most accurate data available by displaying from newest to oldest updated records. The key to this is that all workstations are sending their packets at the same interval. Therefore, packets should always arrive in the same order. This however is not exactly true due to the CSMA random back off period. I do believe though, that overall this does hold approximately true and that more often than not the next packet is the packet to update.

The record to display begins at the most recently updated packet and works back through the list to that record. Thus, during display, updates can occur on records which are yet to be displayed. This allows the most up to date display possible. Any record displayed three times without being updated is deleted from the list.

B.5.10 TrekTorpList.mesa

This is a linear list of the torpedoes in the local ship's system including their current display and move information. Torpedoes fired within a range which gives it any chance of being seen by the ship are recorded in this list. Once every game interval this list is searched. Every torpedo is erased, moved, tested for expiration or a collision. If the torpedo has not expired and no collision occurs, then it is redisplayed. If the torpedo has expired or a collision is detected, then it is not redisplayed and its record is deleted. If a collision with a starbase is detected, the starbase's owner is informed.

B.6 TrekDisplay.mesa

B.6.1 Static data

This module defines the Trek game displays. It contains all of the constant data which define the windows, subwindows, and regions within them. For this information, box dimensions and distances are in pixels. This module also contains the display sizes, dimensions, for objects in all views (long range scan, short range scan, command window, status window, galactic map, message window, and the remote scan window).

B.6.2 In line procedures

This module also defines several in line procedures. These procedures provide common display functions used by Trek. They are in line functions defined here because they typically perform display related functions of one or two lines which are best handled as a single call, but do not justify the time to actually go to a new procedure in a new module and return.

One example of the procedures in this module are AdjX and AdjY which compute shifts in X and Y coordinates given the current values and the angle and distance vector information. These are simple

mathematical functions which could be typed at the point they are used but for appearance the adjust function is called returning a value.

Another example are display procedures which take, as input my display object data structure, disassemble it, and appropriately call the system display functions. The systems functions could be used directly in the source instead of these calls, but again the lines would be long dereferencing through the display object. This interface allows the source to say DisplayObject [object] and have all the work of splitting out the data in object and displaying it handled automatically. This without the cost of an extra procedure call.

B.6.3 Significant data structures

The important types defined in this module include the display object mentioned above, and the enumerated types used to create the various display windows. The display object contains the source bitmap address for the objects display. It also contains the objects width and the objects half width. These values are supplied directly to the systems display procedures or used to create values which are passed to the display procedures.

B.6.4 Significant global variables and Procedures

This interface provides several global variables and procedures used for the display. The global variables include values which are set in the galactic map and the main data element defined by TrekOps. These are provided globally here because TrekDisplay is implemented in several modules. Specifically one module is used in constructing all of the windows while other modules contain the callback procedures for handling window actions and displays. The callback procedure modules do not have ready access to data from the module which constructed the windows. Therefore, while the module which constructed the windows supplies the actual storage for these values, they need to be

referenced by other modules which get them globally through the TrekDisplay interface. The procedures defined here will be discussed with their implementation modules below.

B.6.5 TrekDisplayImplA.mesa

This is the first of nine display implementation modules. It provides for construction of all the Trek game windows. It also contains the procedures which support registration and implementation of the executive Trek operations. Further, it provides the backing store for the global variables used in all display activities. Init and Entry from figure B.2 are implemented here.

B.6.6 TrekDisplayImplB.mesa

This display module supplies the display call back procedures and their supporting routines. Its procedures are supplied by the widow creation procedures in TrekDisplayImplA to the Tajo windowing system. They are called by the system when the window or some portion of it needs to be refreshed.

B.6.7 TrekDisplayImplC.mesa

TrekDisplayImplC implements the game's command and change call back procedures. It has a main entry procedure, CmdProc, which is graphically shown in figure B.3. This procedure is supplied by TrekDisplayImplA to all of Trek's command windows as the call back for the mouse selectable commands and boolean items. Tajo supplies CmdProc with the window and action which initiated the call. Based on this information the procedure calls appropriate supporting procedures to perform the desired function. The supporting procedures start / stop the game, fire phasers / torpedoes, raise / lower shields, deploy starbases, send messages, and switch the battle computer, warp engines, and cloak on / off. Some are also called by TrekDisplayImplN directly and will be readdressed in that section. The phaser and torpedo weapons fire procedural flow is shown in figure B.7.

B.6.8 TrekDisplayImplD.mesa

This module implements the display data for objects other than stellar bodies. The bitmaps for stellar bodies are implemented in the TrekBodiesList data base. The bitmaps for ships, bases, torpedoes, explosions, and shields, both long and short range scan views are all allocated here as variables. This allows them to be accessed by pointers for the display routines. The procedures exported from this module accept data to calculate which bitmap is required and return the appropriate display object. The display object contains a pointer to the requested bitmap with the information to display it.

B.6.9 TrekDisplayImplE.mesa

This module is a monitor created specifically to govern phaser fire. It is graphically shown in figure B.7. Torpedo fire is handled the same way, but uses the monitor in TrekDisplayImplC. To prevent torpedo reloads from blocking phaser fire and phaser recharges from blocking torpedo fire, the two systems each have their own monitor. Two procedures are exported by this module. They are PhaserFire and ClearPhaserQueue.

The phaser fire command is called from TrekDisplayImplC or TrekDisplayImplN to queue a call to fire the ships phasers. This routine limits the fire queue to 5 shots. Each shot is queued by the monitor to PhaserFiring. PhaserFiring pops shots off the queue, generates the packet which represents the phaser fire, forks the base test for phaser fire, and reenergizes the phaser. As mentioned earlier, the firing ship is responsible to test for phaser fire hitting a base. The phaser reenergization pauses for the ships reenergization period while in the monitor. This prevents another shot from being processed until the reenergization is finished. ClearPhaserQueue is called when the game is stopping. It stops phaser packet transmissions and flushes the queue so no more packets will be generated.

B.6.10 TrekDisplayImpl.mesa

This module is a monitor which implements the heart of the Trek game. It exports two procedures. One procedure starts the game's display activity and one stops it. Operation of the display involves two processes. They are the WakeUp process and the DoIt process graphically shown in figure B.5. WakeUp broadcasts a signal every game interval for the DoIt process to wake up and update the game and display. Implementing this with only the DoIt process would have the game interval plus the game / display update time which is variable.

DoIt performs the following activities to update the game and display. First it charges energy for active ship services such as cloak, shields, acceleration, and life support. Then it computes the ships new position. This is done now so that the following display events do not have to wait for computations. Next the local ship is tested for any collisions. This is done now to avoid any further work if the ship is destroyed. Now, the display sequence begins. The variable displays for ship status are erased if they have changed. The Ships status packet is updated from the local ships information computed earlier. The status packet will now have the latest data for transmission to other ships and for update of displays which are driven by it. All objects in the display list are next erased. The local ships display is erased and the solar system is shifted to represent local ship movement. The local ship is then redrawn. The call back procedure from the shift automatically adds any stellar bodies and star bases made visible by the shift. The ship data base is now inspected for visible ships to be drawn in their updated locations. The torpedoes are updated; moving to their new positions and testing for collision before being displayed. Finally, any new or modified bases are displayed appropriately from the base display list.

The ship and torpedo displays are forked processes. The base display runs in the DoIt process. All must complete before the DoIt process sleeps waiting for the next wake up. By being processes, if any of them hang up waiting for the data base monitor to come free while it updates the data base, the

other process may continue to display information. They are spawned ship, torpedo, and then bases run in the local process. They are joined back torpedo then ship after the base update completes. The ship process is expected to be the longest; more of them and more update interruptions. The torpedo process is next; sometimes there are a lot of torpedoes, but often there are none. The base display is expected to be the shortest; bases are rarely deployed and even rarer does their display change. Thus, the processes are spawned and joined for optimal efficiency.

B.6.11 TrekDisplayImplM.mesa

This is the game's control monitor. It regulates game start, stop, die, warp, and base transport. These procedures start, stop, and reset game monitors and data bases. If not properly regulated the game can end up in deadlock or other undesirable states. The primary export of this module is GameCntrl. It takes as input the desired operation. It is monitored to insure that the supplied operation is allowed given the game's current state. If the action is allowed, the new state is recorded and the action performed. If the action is not allowed but will be, the procedure waits for an appropriate broadcast before changing state and performing the operation. If the action is inappropriate given the game's current state and will not be appropriate, the action is ignored. An example of this is multiple reasons for dieing, out of energy and to much damage simultaneously. The first death initiates the sequence. Subsequent death calls while dieing are ignored. By the time the death sequence is over, it has reset the damage control system and energy system so no further calls will be made until actual death of the now resurrected ship. The Start procedure from this module is graphically shown in figure B.2.

B.6.12 TrekDisplayImplN.mesa

This module implements the game's notifier. The custom Trek window TIP tables are created here and attached to the appropriate subwindows along with the notify call back procedure in figure B.3. The mouse commands which are position and button dependent, not command or boolean objects,

are sent to the notifier call back procedure. The Tajo notifier calls this procedure with mouse coordinates and atomic commands also defined in this module and referenced in the TIP tables. Based on this information the command is interpreted and acted upon. This typically results in a call to TrekDisplayImplC to perform the actual command.

B.6.13 TrekDisplayImplP.mesa

This module is a monitor specifically designed to regulate phaser fire and scan display shift. Because of the way phaser fire is implemented the short range scan display may not be shifted during phaser display. Therefore, the phaser and local ship / shield display shift procedures are monitored to block one another.

The shield display also makes use of the monitor to protect against simultaneous changes of the shields. Damage, battle computer, and multiple command entry paths can result in erasure and display of the shields. Therefore, the monitor is employed to insure that only one change occurs at a time. All other changes wait until the monitor is free and then make their change. If this were not employed, it would be possible for damage to erase a medium shield and display a low shield while a raise shield command erased the medium shield and displayed a high shield. In the end the shield setting and display would not necessarily be the same. Thus it is monitored to block such an occurrence.

B.7 TrekCollision.mesa

B.7.1 Static data

This module defines the interface between the game and the ships damage control. It provides constant values which define energy drain for special ship operations (shields, phasers, cloak, ...). It also defines the damage severity levels (Slight, Moderate, and Severe).

B.7.2 Significant data structures

This interface defines the enumerated list of ship systems which may be damaged. The list is used to randomly assign damage, to list the damage, and to make the damage values available indexed by ship system.

B.7.3 Significant global variables and Procedures

To make the damage values available, indexed by ship system, this interface provides a global array of system damage indexed by the ship system enumeration. The module also defines an interface between TrekCollisionImpl and the rest of the Trek game. TrekCollisionImpl implements the variables and procedures defined by this module. The interface provides for initiating, terminating, and resetting the game's damage control system. Also provided are procedures for assigning damage to both the local ship and its starbases. Further, this interface provides some collision detection software for generic vectors and phaser specific vectors.

B.7.4 TrekCollisionImpl.mesa

This module implements the ship's damage control system. Any collisions which are detected for the ship or its bases use procedures implemented here to assess damage. This module also implements the data base for damaged systems and provides access to that information so that the game may determine if a system is functioning, operating incorrectly, or inoperative all together.

This module is implemented as a monitor which has the damage control process, shown in figure B.6, spawned at game start up. This process automatically performs ship repairs. When the ship is docked it escalates the repairs to represent assistance and raw materials from the star base. Also, when docked this module consumes base energy to resupply the local ship's energy and weapons

systems. If the base has resources and has constructed new bases, they too are picked up by this module and added to the ship's stores.

B.8 TrekCommon.mesa

TrekCommon supplies no constants nor does it define any data structures. It primarily provides in line procedures. As with some of the TrekDisplay in line procedures, these procedures are typically short mathematical procedures. They perform real and integer functions to calculate distance, convert between game units of measurement, normalize coordinates and angles, translate race into specific object types (base, ship, torpedo, phaser). All of these are simple operations which are done neatly with a procedure call, but could be done in the calling procedure.

This interface does define one procedure which is implemented by TrekCollisionImpl. It is defined here because of its common usage. The procedure is the Random number generator. It accepts as input a positive number. It returns a value from 0 to the supplied value less one. It was initially a local procedure to the collision module, but exported through TrekCommon when other procedures such as ship position initialization needed it.

Appendix C

User Guide

Typing help trek in the XDE executive will generate the following documentation text without the graphics and font changes. The diagrams in this section are screen copies of an actual game. They are only available in this documentation format and not in the executives help facility which only displays vanilla text.

Trek User's Guide

Environments - Trek is designed to run in the Tajo Environment. The initial release is for the 14.0 level of software. It is able to run within the VP tajo hack. The game is played over the local ethernet. Each workstation is a single ship. The game does not go over the internet.

Necessary Software - Trek.bcd and DisplayImpl.bcd for the Tajo boot file environment. Trek.bcd only for the Tajo in VP hack. DisplayImpl is a software package provided by mesa for manipulating the display. Viewpoint includes its own version of this package, Tajo does not. DisplayImpl is used by other tools in Tajo and once activated it is available to all applications so it may not be necessary for the user to load this. However, it is necessary that it be on the current search path so that it may be loaded automatically if it is needed.

Operation - Run Trek.bcd in the Executive window. This will bring up a window for selecting game parameters and starting / stopping the game. It will also register Trek with the Executive Help and Unload commands. Help Trek will type this document in the Executive window (without the graphics and font changes presented here). Unload Trek will remove Trek from the system. To automatically position Trek windows use the following fields in your user.cm with the desired WindowBox and TinyPlace values: TrekSRS, TrekMsg, TrekCmd, TrekStatus, TrekGM, TrekRemoteScan, TrekLRS, and TrekEntry. Specific values for fields in the TrekEntry window may be set as in the following sample user.cm slice.

[TrekSRS]
WindowBox: [x: 0, y: 322, w: 514, h: 526]

[TrekMsg]
WindowBox: [x: 0, y: 288, w: 514, h: 46]

[TrekCmd]
WindowBox: [x: 258, y: 159, w: 256, h: 141]

[TrekStatus]
WindowBox: [x: 258, y: 30, w: 256, h: 141]

[TrekGM]
WindowBox: [x: 513, y: 464, w: 258, h: 270]

[TrekRemoteScan]
WindowBox: [x: 513, y: 206, w: 258, h: 270]

[TrekLRS]
WindowBox: [x: 0, y: 30, w: 258, h: 270]

[TrekEntry]
WindowBox: [x: 513, y: 30, w: 512, h: 188]
ID: yourHandle
Start: TRUE
Race: Klingon
System: Sol
Gravity: TRUE

To begin play enter desired values and make desired selections for your ship in the Trek window. The following shows the function of the fields presented.

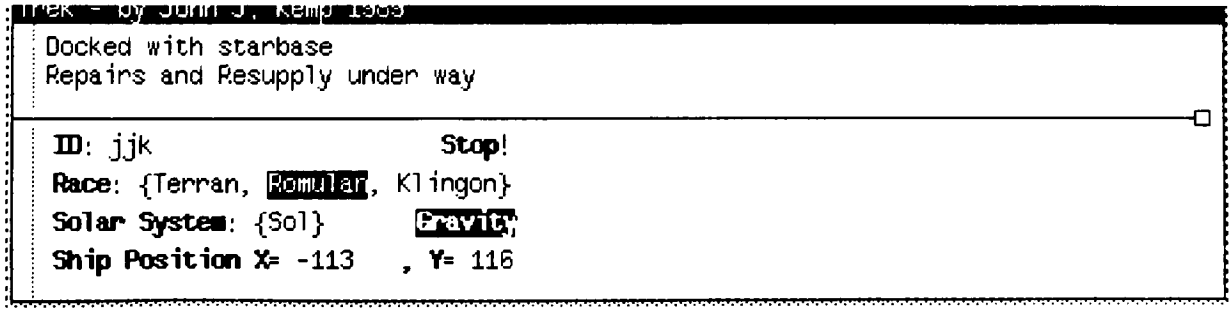


Figure C.1 Trek parameter entry window.

ID: = Enter the identity by which you wish to be known to others while playing the game.

Race: = Select the ship type you wish to use. The ships are defined at the end of this document. Selection is made by pointing with the mouse at the desired race.

Gravity = When highlighted, the default, gravity is on for your ship which will be attracted to the center of the solar system. When not highlighted, gravity is off and the novice player will not have to worry about being drawn into solar system's center.

Solar System: = Chord over this field and select the system you wish to start in. Selection is made by simultaneously pressing both mouse buttons over the Solar System field, moving the mouse so the desired system is highlighted and releasing the buttons.

Ship Position = Read Only. The ship's position is displayed here in graph coordinates for the Solar System above.

Select Start! to begin the game.

Once Start is pressed the game begins; the start command is replaced with Stop and the race, gravity, and solar system fields become read only. Six windows are then presented in addition to the Trek window. They are: Long Range Scan, Short Range Scan, Status, Command, Message, and Galactic Map. Each of these windows is described below.

Select Stop! to exit the game.

Long Range Scan Window - This window shows a 256 × 256 view with the local ship in the center. 1 game unit of space = 1 pixel, but bodies, ships, and bases are displayed at double their actual size (1 game unit = 2 pixels). In this window, the left mouse button fires phasers, the right mouse button fires torpedoes. Torpedoes are not displayed in this window.

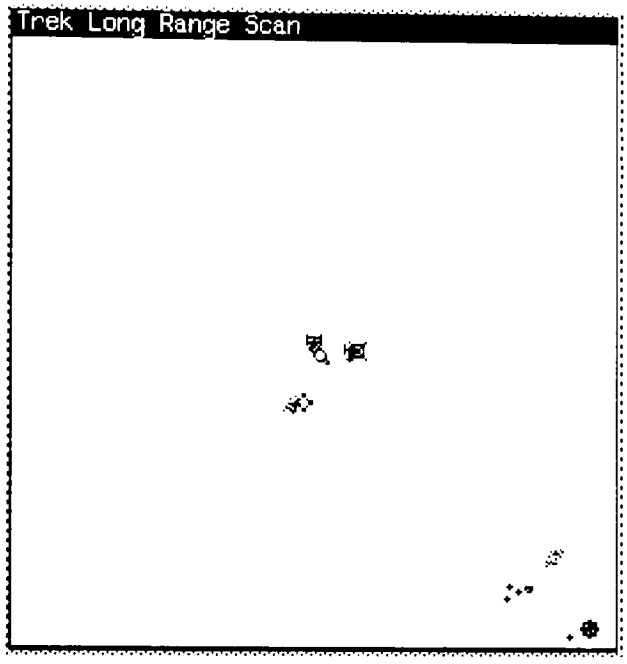


Figure C.2 Long Range Scan window.

Short Range Scan Window - This window shows a 512 × 512 view of a 128 × 128 square around the center of the long range scan. This is a 4 × view of the game where 1/4 of a game unit = 1 pixel. In this window the left mouse button fires phasers, the right mouse button fires torpedoes.

Status Window - This window has 5 subparts (velocity display, energy display, StarDate, Damage, and Base & Torpedo count indicators).

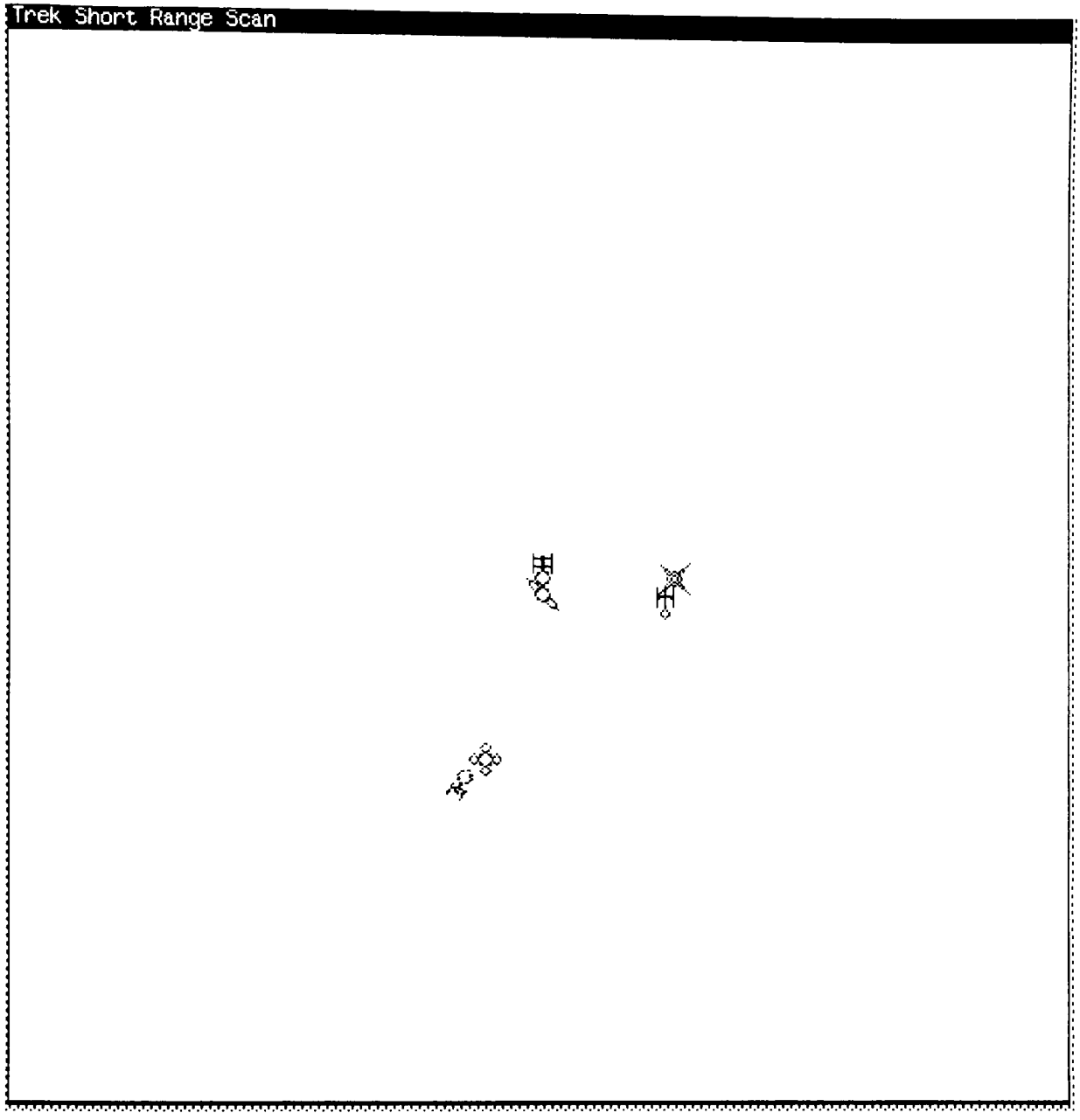


Figure C.3 Short Range Scan Window.

Velocity - displays a vector indicating ship direction and speed. Ship speed is displayed as a percentage of maximum. The outer circle represents 100% of the ships maximum speed.

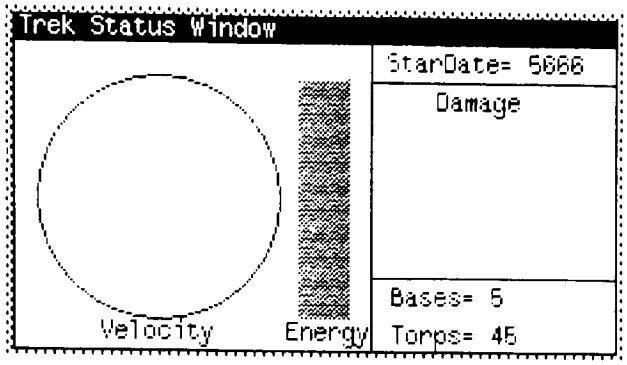


Figure C.4 Status Window.

Energy - indicates the ships energy reserves.

StarDate - is the number of game intervals the game has been played.

Damage - displays any damage sustained by the ship and its severity. The damaged systems are shown in order (severe, moderate, slight). If there is insufficient room to display all of the systems, the lower severities are not shown.

Base & Torpedo count indicators - show the number of bases and torpedoes available to the ship. These values are decremented when torpedoes are fired and bases deployed. They are initially set to their maximum value for the given ship and resupplied when docked with a star base capable of resupply activities. Star bases must be near a body in order to produce more star bases.

Command Window - This window has four sections (Direction, Accel, systems, and commands).

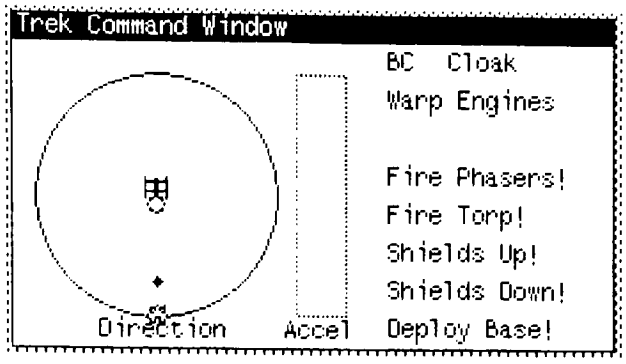


Figure C.5 Command Window.

Direction - is the heart of the game. In this section the ships direction, fire control, and shields are controlled and displayed. It has 4 zones with the following actions for the left and right mouse buttons in each zone:

Zone	Description of zone	Left button	Right button
Ship & Torpedo Control	Outer zone on circle	Position ship direction target	Fire torpedo
Phaser Control	Just inside circle	Position phaser target	Fire phasers
Shield Control	Around the ship in circles center	Raise shield	Lower shield
Fire Control	Ship in center of circle	Fire phasers	Fire torpedo

Table C.6 Command Window mouse action description.

Accel - shows the current acceleration as a percentage of maximum acceleration for the ship. It is set by pointing at the desired level of acceleration with the mouse and pressing the left mouse button. When the ship reaches maximum velocity in the desired direction the acceleration is automatically set to 0.

Systems - contains the BC (Battle Computer), Warp Engines, and for Romulans the Cloak system indicators. These fields act as booleans and are turned on (inverted), or off (normal) by pointing at the system and pressing the left mouse button.

BC - on will target phasers and torpedoes at the closest ship directing the ship into combat. It also compensates for gravity and automatically maintains the shields at Front = High, Left = Low, Right = Low, and Rear = None..

Warp Engines - on will, when the ship is at its maximum velocity, attempt to warp to a system in the direction of the ships velocity. If a system is not available along the velocity vector, the ship warps within the local system. The warp will drop the ships shields and leave the ship unprotected for 1 game interval before taking the ship out of the system.

Cloak - on will eliminate the long range scan windows functionality, and remove the ship from all other long range scans. It will however retain a ghost image of the long range scan and plot this as the ship moves. This allows the player to see how his ship is maneuvering with regard to its last known view. This system is only available to Romulans.

Command - contains five commands (Fire Phasers, Fire Torp, Shields Up, Shields Down, and Deploy Base). The commands are triggered by pointing at the command and pressing the left mouse button. The fire commands are alternatives to the mouse actions in this and other windows. Through any means of firing a weapon there is a limit of 5 queued requests. This will prevent extraneous firings in the heat of battle from wasting torpedoes and energy. The shield commands are a means of quickly raising or lowering all shields 1 level. The Deploy Base command deposits a base, if one is available, in the direction of the phaser control.

Message Window - The Message window has four areas (incoming message, out going message, Send command, and ship count indicators.

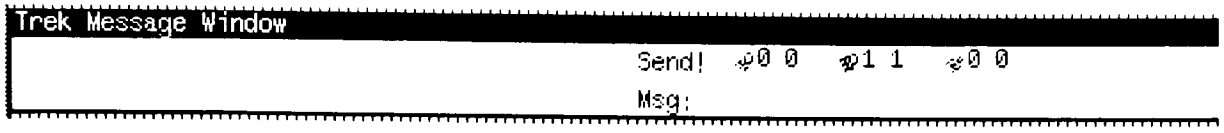


Figure C.7 Message Window.

Incoming message - displays messages received from other ships with the ships id and system. This is the left half of the message window.

Outgoing message (MSG:) - is where messages to be sent are typed. Typing a carriage return in the message will send what has been typed to that point and reset the field.

Send! command - An alternative way of sending the outgoing message is to point at the send command and press the left mouse button.

Ship count indicators - displays ship icons followed by two numbers. For each of the ship types shown, the following numbers represent the number of that ship type in the game and local system respectively.

Galactic Map Window - This window has 3 sections (systems, base, and command).

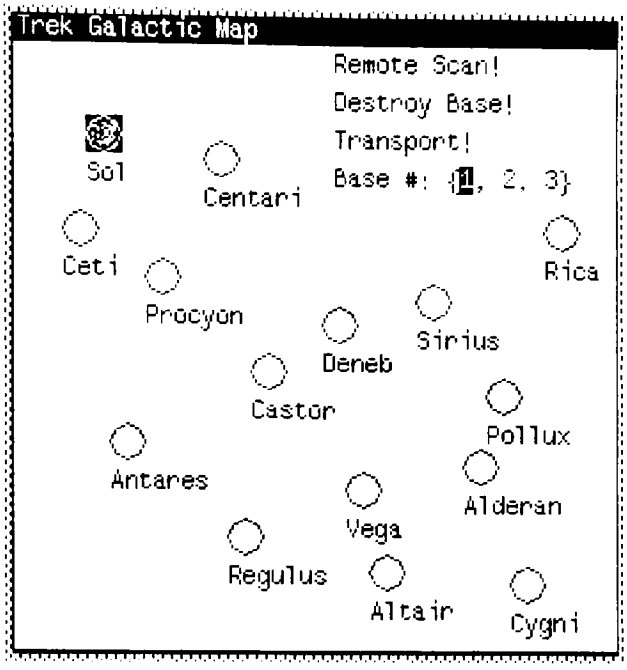


Figure C.8 Galactic Map Window.

Systems - show the known galaxy. Each system is displayed as a circle. The center of this circle is inverted if a base exists in the system. For base commands a system must be selected by pointing to it and pressing the left mouse button. The selected system is indicated by being inverted. The system the ship is currently in has a ship displayed over it.

8ase - offers choices 1 - 3 for the base in the system. If multiple bases are present in a system this allows choosing between them. If multiple bases are in a system and 1 is destroyed its number will no longer respond and another needs to be tried.

Command - offers three base operations (Remote scan, Destroy base, Transport). These are performed by pointing to the desired command and pressing the left mouse button.

Remote scan - will make a scan window available which shows a long scan view around the indicated base. Remote scan from another base will switch the windows view to that base, and remote scan from a non existent base will stop the remote scan display.

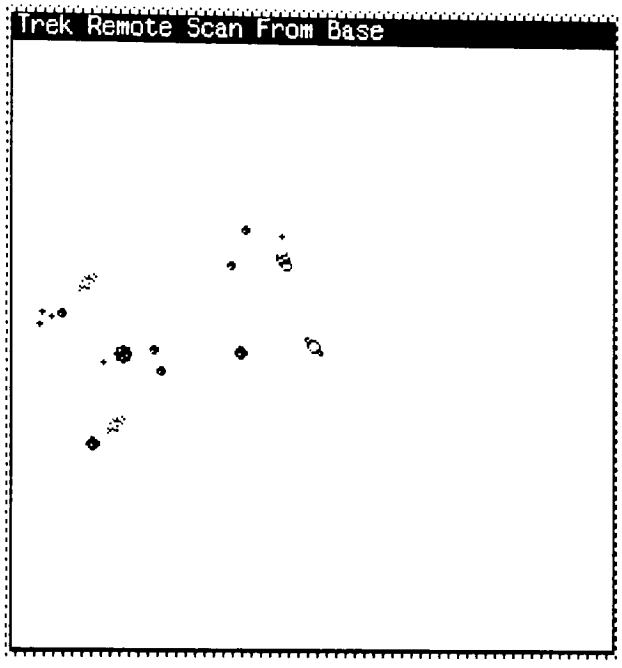


Figure C.9 Remote Scan Window.

Destroy base - will cause the indicated base to explode.

Transport - will, if the base has sufficient energy, transport the ship to the base such that the base is in the direction of the phaser target. Like warp, the ship's shields are dropped for 1 game interval before the ship leaves the system.

Ship Damage - Damage is sustained when an object collides with another object or is hit with an energy blast or phaser weapon. The local ship is charged damage randomly to several systems. The damage is computed from a minimum collision value for the item plus a variable component which

for objects is the impact velocity and for energy (phaser and explosion) is based on distance from the energy source. Damage is reduced if there are shields on the side of the impact based on the intensity of the shield. When the damage level for a given system reaches moderate the system fails and ceases to operate. This is implemented differently for each system. Torpedoes and phasers are affected at any level of damage each point of damage is 1 millisecond additional delay in reload / reenergization time. They cease to work at the moderate level.

Base Damage - Bases suffer damage the same as ships. They may also be captured. A failed capture attempt results in destruction of the ship attempting to capture the base. If the ship captures a base but already has 3 bases in the system the base is disassembled.

Ship characteristics - Each race has special ship characteristics listed below:

Terran - Overall this is the average ship. It has both defensive and offensive pluses, but no distinguishing feature like the Klingon phaser or Romulan cloak. It has 2 medium phaser streaks and a fair number of torpedoes, and is generally average in all characteristics.

Maximum Damage	1200
Bases	4
phaser streaks	2
phaser range / power	30
Torpedoes	32
Torpedo Load Time	1200
Phaser Energize Time	300
Energy Recovery Rate	8
Maximum Shield Level	3
Maximum Rotation	30
Maximum Speed	2.5
Maximum Acceleration	.3
Maximum Accel. Change	.24

Romulan - This ship is sturdier than the others defensively, but weaker offensively. Unlike the other ships it offers a cloak command which renders the ship invisible to others except in their short range scan. It however removes the cloaked ships long range scan ability. The ships phasers are short and weak, but cover a wide area. Its strength is in its large number of torpedoes allowing it to attack at a distance, and its strong defenses, higher shield ability, allowing it to sustain several hits before being destroyed.

Maximum Damage	1400

Bases	6
phaser streaks	3
phaser range / power	25
Torpedoes	45
Torpedo Load Time	1500
Phaser Energize Time	350
Energy Recovery Rate	12
Maximum Shield Level	4
Maximum Rotation	40
Maximum Speed	2
Maximum Acceleration	.2
Maximum Accel. Change	.3

Klingon - This ship is weak defensively, but powerful offensively. It is the more challenging ship, offering a powerful, single streak phaser and few torpedoes which require all offensive activities to be precise, and its weak defensive abilities require that the offensive actions be taken quickly.

Maximum Damage	1200
Bases	3
phaser streaks	1
phaser range / power	35
Torpedoes	15
Torpedo Load Time	1000
Phaser Energize Time	250
Energy Recovery Rate	8
Maximum Shield Level	3
Maximum Rotation	20
Maximum Speed	3

Maximum Acceleration	.3
Maximum Accel. Change	.3

References

1 Atkinson, Russ. 1987. Tank Documantation. Palo Alto, California: Palo Alto Research Center, Xerox Corp.

2 Ball, Eugene., etal. 1981. Trek BCPL Source Code. USA: Xerox Corp.

3 Curry, James E. & PARC staff. 1979. BCPL Reference Manual. Palo Alto, California: Palo Alto Research Center, Xerox Corp.

4 Henning, Gerhard. 1988. Blub Server Mesa source code. Munich, Germany: Siemans Corp.

5 Jell, Thomas. 1988. BlubWorkstation Mesa source code. Munich, Germany: Siemans Corp.

6 Jell, Thomas., Gerhard Henning, etal. 1988. Blub Documentation. Munich, Germany: Siemans Corp.

7 Maxwell, John and Russ Atkinson. 1987. Tank CEDAR source code. Palo Alto, California: Palo Alto Research Center, Xerox Corp.

8 Xerox. 1984. Mesa Language Manual. USA: Xerox Corp.

9 Xerox. 1984. Mesa Programmer's Manual. USA: Xerox Corp.

10 Xerox. 1986. Pilot Programmer's Manual. USA: Xerox Corp.

11 Xerox. 1986. XDE User's Guide. USA: Xerox Corp.

12 Yamamoto, Bryan. 1984. MazeWar Mesa source code. USA: Xerox Corp.